全美经典
学习指导系列

数据结构
习题与解答
Java语言描述

DATA STRUCTURES WITH JAVA

最佳的复习资料，实用的辅助教材

与国外高校计算机水平保持同步

为考研和出国深造奠定坚实基础

John R.Hubbard 著
周斌 刘亚萍 等译

全球销售超过3000万册！

机械工业出版社 China Machine Press
中信出版社

全美经典学习指导系列是一套快捷有效的学习指南，该套丛书针对各专业的技术重点提供了数百个实例、习题及答案。通过这些实战练习，不但可以洞悉各门技术精髓，而且能够使考试成绩大幅攀升，更会助你与国外大学生的计算机水平看齐，为将来考研或出国深造奠定坚实基础。

全美经典学习指导系列深得高校学生的喜爱。由于有了这套丛书，在历年的专业考试中，成千上万的学生获得了优异成绩。

想成为一名优等生吗？——请选择全美经典学习指导系列！

如果时间不够却想成编程人，这本书可以助你：

- 通过具体范例解决疑难问题
- 考前快速强化
- 迅速找到答案
- 快捷而高效地学习
- 迅速掌握技术重点，无需翻阅冗长的教科书

全美经典学习指导系列以方便快捷的形式提供了考生需要了解的信息，同时不致使你淹没在不必要的细节当中。另外，还可以通过大量的编程练习来测试所学的技巧。该丛书可以与任何教材配合使用，使学生们能够根据各自的进度来学习，从而获得事半功倍的效果！**全美经典学习指导系列**的内容系统而完备，是毕业考试和专业考试的理想参考书。

本书包括：

- 使用目前最热门的语言，阐述计算机科学－Ⅰ，Ⅱ
- 计算机专业问题的详细例释
- 涵盖计算机科学高级课程的第二部分
- 大量的例子和习题解答将帮助你通过 Java 语言掌握数据结构

如果想获得优异成绩并且能够全面掌握数据结构的原理，本书是不可或缺的最佳辅导老师

C++ 编程习题与解答

C++ 编程习题与解答（英文版·第二版）

Java 编程习题与解答

Java 编程习题与解答（英文版）

SQL 编程习题与解答

Visual Basic 编程习题与解答

操作系统习题与解答（英文版）

关系数据库习题与解答

关系数据库习题与解答（英文版）

计算机导论习题与解答

计算机导论习题与解答（英文版）

计算机体系结构习题与解答（英文版）

计算机图形学习题与解答

计算机图形学习题与解答（英文版·第二版）

计算机网络习题与解答

软件工程习题与解答

数据结构习题与解答

——对应 C 语言描述

数据结构习题与解答

——对应 Java 语言描述（英文版）

数据结构习题与解答

——对应 C++ 语言描述（英文版）

ISBN 7-111-10829-9/TP·2517

定价：30.00 元

ISBN 7-111-10829-9

9 787111 108290

网上购书：www.china-pub.com

北京市西城区百万庄南街 1 号 100037

购书热线：(010)68995259, 8006100280

(北京地区)

总编信箱：chiefeditor@hzbook.com

标准分享网 www.bzfxw.com 免费下载

全美经典
学习指导系列

数据结构习题与解答

Java语言描述

DATA STRUCTURES WITH JAVA

John R. Hubbard 著
阳国贵 周斌 刘亚萍 等译

机械工业出版社 China Machine Press
中信出版社 CITIC PUBLISHING HOUSE

本书介绍了网络环境下体现软件工程和面向对象理念的JAVA编程语言，讲述了它的面向对象、分布式、可移植性等诸多特点以及如何使用JAVA语言来实现各种主要的数据结构。全书使用了大量的源程序，从一个程序员的视角重新诠释了数据结构以及JAVA实现，是一本不可多得的好书。

John R. Hubbard: Data Structures with JAVA.

Copyright © 2001 by The McGraw-Hill Companies, Inc.

Original language published by The McGraw-Hill Companies, Inc. All Rights reserved. No part of this publication may be reproduced or distributed in any means, or stored in a database or retrieval system, without the prior written permission of the publisher.

Simplified Chinese translation edition jointly published by McGraw-Hill Education (Asia) Co. and China Machine Press & CITIC Publishing House.

本书中文简体字翻译版由机械工业出版社、中信出版社和美国麦格劳-希尔教育(亚洲)出版公司合作出版。未经出版者预先书面许可，不得以任何方式复制或抄袭本书的任何部分。

本书封面贴有McGraw-Hill公司防伪标签，无标签者不得销售。

版权所有，侵权必究。

本书版权登记号：图字：01-2001-5201

图书在版编目（CIP）数据

数据结构习题与解答：Java语言描述/（美）哈伯德（Hubbard, J. R.）著；杨国贵等译. -北京：机械工业出版社，2002.8

（全美经典学习指导系列）

书名原文：Data Structures with Java

ISBN 7-111-10829-9

Ⅰ.数… Ⅱ.①哈… ②杨… Ⅲ.①语数结构－解题－解题 ②JAVA语言－程序设计－解题 Ⅳ.TP311-44

中国版本图书馆CIP数据核字（2002）第062687号

机械工业出版社（北京市西城区百万庄大街22号 邮政编码100037）

责任编辑：华章

北京忠信诚印刷厂印刷·新华书店北京发行所发行

2002年8月第1版第1次印刷

787mm×1092mm 1/16·27.5印张

印数：0 001-5 000册

定价：39.00元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换

标准分享网 www.bzfxw.com 免费下载

作 者 序

与所有 Schaum 系列丛书中的其他书籍一样，本书主要是为自学者编写的，特别适合与数据结构的课程一起使用，同时也可以作为单独的学习用书或参考书。

书中包括 200 多个例题、习题及习题解答。作者坚信数据结构原理可以通过精心设计并有完整解释的例子开始学习，本书正体现了这一观点。

书中所有例题和习题的源代码均可从http://projectEuclid.net/schaums/dswj/ 或其镜像站点 http://www.richmond.edu/~hubbard/schaums/dswj/进行下载，对本书的任何修正或补充部分也可从这些站点获取。

感谢对书稿提出过改正意见的所有朋友、同事、同学以及 McGraw-Hill 出版社的工作人员。在此，还要特别感谢我的妻子及同事 Anita H.Hubbard，本书得益于她的诸多建议、鼓励和创造性思想，并采用了许多由她提供的习题。

目　录

第 1 章　Java 回顾 …… 1
1.1　面向对象的程序设计 …… 1
1.2　程序设计语言 Java …… 1
1.3　变量和对象 …… 2
1.4　基本类型 …… 3
1.5　流程控制 …… 5
1.6　类 …… 8
1.7　修饰符 …… 11
1.8　String 类 …… 12
1.9　Math 类 …… 16
第 2 章　数组回顾 …… 29
2.1　数组的特性 …… 29
2.2　数组复制 …… 31
2.3　Arrays 类 …… 32
2.4　顺序查找算法 …… 37
2.5　折半查找算法 …… 39
2.6　Vector 类 …… 42
第 3 章　Java 高级特性 …… 69
3.1　继承 …… 69
3.2　多态性 …… 70
3.3　类型转换 …… 72
3.4　Object 类 …… 75
3.5　抽象类 …… 78
3.6　接口 …… 81
3.7　包 …… 85
3.8　异常处理 …… 85
第 4 章　递归 …… 93
4.1　递归基和递归体 …… 94
4.2　跟踪递归调用 …… 95
4.3　递归折半查找算法 …… 97
4.4　二项式系数 …… 99
4.5　欧几里德算法 …… 100

标准分享网 www.bzfxw.com 免费下载

4.6 正确性的归纳证明 …… 101
4.7 递归算法的复杂性分析 …… 101
4.8 动态规划 …… 102
4.9 汉内塔 …… 103
4.10 互递归 …… 105
第5章 汇集 …… 119
5.1 Java汇集框架 …… 119
5.2 Collection接口 …… 120
5.3 AbstractCollection类 …… 120
5.4 Bag类 …… 122
5.5 Iterator接口 …… 130
第6章 栈 …… 137
6.1 Java中的Stack类 …… 137
6.2 栈的应用 …… 141
6.3 消去递归 …… 144
第7章 队列 …… 153
7.1 队列框架 …… 153
7.2 顺序实现 …… 156
7.3 链接实现 …… 160
7.4 队列应用 …… 162
第8章 线性表 …… 178
8.1 java.util.List接口 …… 178
8.2 java.util.List接口的实现 …… 180
8.3 AbstractList与AbstractSequentialList类 …… 181
8.4 线性表迭代器 …… 182
8.5 ArrayList类 …… 184
8.6 LinkedList类 …… 186
8.7 独立线性表迭代器 …… 197
第9章 树 …… 204
9.1 树的定义 …… 205
9.2 决策树与迁移图 …… 207
9.3 有序树 …… 210
9.4 有序树的树遍历算法 …… 211
第10章 二叉树 …… 220
10.1 定义 …… 220
10.2 二叉树的计数 …… 221

10.3 满二叉树 …… 222
10.4 相同、相等和同构 …… 223
10.5 完全二叉树 …… 225
10.6 二叉树遍历算法 …… 227
10.7 表达式树 …… 229
10.8 二叉树类 BinaryTree …… 231
10.9 遍历算法的实现 …… 237
10.10 森林 …… 240
第 11 章 搜索树 …… 252
11.1 多路搜索树 …… 252
11.2 B-树 …… 254
11.3 二叉搜索树 …… 257
11.4 二叉搜索树的性能特点 …… 258
11.5 AVL 树 …… 259
11.6 类 AVLTree …… 260
第 12 章 堆和优先队列 …… 268
12.1 堆 …… 268
12.2 自然映射 …… 268
12.3 堆的插入 …… 269
12.4 堆的删除 …… 270
12.5 PriorityQueue 类 …… 271
12.6 Java 的 Comparator 接口 …… 272
12.7 优先队列的一个直接实现 …… 275
第 13 章 排序 …… 289
13.1 Java 中的 Arrays.sort()方法 …… 289
13.2 冒泡排序 …… 290
13.3 选择排序 …… 292
13.4 插入排序 …… 293
13.5 SHELL 排序 …… 294
13.6 归并排序 …… 296
13.7 快速排序 …… 299
13.8 堆排序 …… 302
13.9 比较排序的速度限制 …… 306
13.10 基数排序 …… 307
13.11 吊桶排序 …… 309
第 14 章 表 …… 325

标准分享网 www.bzfxw.com 免费下载

14.1 Java的Map接口 …… 325
14.2 HashMap类 …… 326
14.3 Java的Hash代码 …… 328
14.4 Hash表 …… 329
14.5 Hash表的性能 …… 331
14.6 冲突消解算法 …… 332
14.7 独立链 …… 335
14.8 应用 …… 336
14.9 TreeMap类 …… 339
第15章 集合 …… 348
15.1 数学集合 …… 348
15.2 Java集合接口 …… 349
15.3 Java AbstractSet 类 …… 349
15.4 Java HashSet类 …… 350
15.5 Java TreeSet类 …… 353
第16章 图 …… 358
16.1 简单图 …… 358
16.2 图的术语 …… 358
16.3 路径与回路 …… 359
16.4 同构图 …… 361
16.5 图的邻接矩阵 …… 363
16.6 图的关联矩阵 …… 364
16.7 图的邻接表 …… 364
16.8 有向图 …… 365
16.9 有向图的路径 …… 367
16.10 加权有向图和加权图 …… 367
16.11 欧拉路径、欧拉回路、哈密顿路径、哈密顿回路 …… 369
16.12 DIJKSTRA算法 …… 369
16.13 图的遍历算法 …… 373
附录A 基础数学知识 …… 393
A.1 下取整与上取整函数 …… 393
A.2 对数 …… 394
A.3 复杂性分类 …… 395
A.4 第一数学归纳法 …… 397
A.5 第二数学归纳法 …… 398
A.6 等比级数 …… 399

A.7　求和公式 …… 400
A.8　调和数 …… 400
A.9　Stirling 公式 …… 402
A.10　斐波那契数 …… 403
A.11　黄金分割 …… 404
A.12　欧几里德算法 …… 405
A.13　Catalan 数 …… 406
附录 B　从 C++到 Java …… 416
附录 C　Java 开发环境 …… 419
C.1　Windows 的命令窗口 …… 419
C.2　Visual Cafe …… 419
附录 D　参考文献 …… 424

标准分享网 www.bzfxw.com 免费下载

第1章 Java回顾

本章概述数据结构设计和实现所用的 Java 语言特征，如要了解更为详尽的内容，可以参见附录D中[Hubband]条目下所列的参考书。

1.1 面向对象的程序设计

Java是一种面向对象的程序设计语言，在该语言中：

- 每个数据项均封装在某个对象中。
- 每条可执行的语句均由某个对象来完成。
- 每个对象均是某个类的一个实例或是一个数组。
- 每个类均在一个单继承的层次结构中定义。

Java中的单继承层次结构是一种树型结构，它的根是Object类（见3.4节）。这种单继承结构似乎使Java成为一种简单的语言，Java 1.3类库就是这样的一个结构，但该类库却包含2130个类并有23 901个成员。

Java的面向对象的特点使得它特别适合于数据结构的设计和实现，它的集合框架（Collections Framework）（见第4章）为程序员构造数据结构提供了一个很好的平台。

1.2 程序设计语言 Java

一个Java程序由一个或多个文本文件组成，其中至少有一个文本文件包含了一个public类（惟一的一个），在该类中含有惟一的一个main方法，其典型用法如下：

```
public static void main(String[ ] args)
```

程序中的每个文件必需包含一个惟一的 public 类或 public 接口定义。对名为 X.java 的文件而言，X 就是该文件中定义为 public 的类名或接口名。每一个这样的文件将被编译为对应的名为 X.class 的类文件。然后，该程序就可通过执行包含 main(String[])方法的 X.class 文件来运行，它将执行 main(String[])方法，该方法中包含了要执行的语句。程序可以命令行的方式来执行，也可在 IDE（集成开发环境）中运行（参见附录C）。

1.3 变量和对象

在任何程序设计语言中，数据的存取均是通过变量来进行的。在 Java 中，一个变量要么是对象的一个引用，要么是 8 种基本类型之一，这些类型间的关系如图 1.1 所示。如果变量是一个引用，那么它的值要么为空（null），要么包含一个实例对象的地址。

每个对象属于一个特定的类。类定义了对象的特性和操作，正如一个基本类型定义了它的变量的特性和操作一样。

一个变量是由说明其类型和初值（可选）的声明来定义的，对象由调用类构造器的 new 操作创建。

例 1.1 创建变量和对象

```
public class Ex0101
{ public static void main(String[] args)
  { boolean flag=true;
    char ch='B';
    short m;
    int n=22;
    float x=3.14159F;
    double y;
    String str;
    String nada=null;
    String country = new String("United States");
    System.out.println("flag = " + flag);
    System.out.println("ch = " + ch);
    System.out.println("n = " + n);
    System.out.println("x = " + x);
    System.out.println("nada = " + nada);
    System.out.println("country = " + country);
  }
}
```

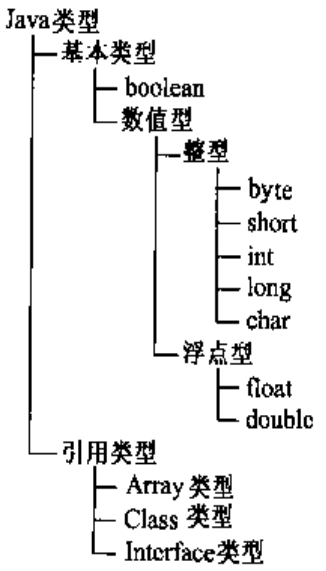


图 1.1 类型间的关系

经编译后运行该程序，其输出将为：

```
flag=true
ch=B
n=22
x=3.14159
nada=null
country=United States
```

在该程序中声明了 9 个变量，随后打印输出了有初值的 6 个变量，这 6 个变量都是基

标准分享网 www.bzfxw.com 免费下载

本类型变量，最后的 3 个变量是指向字符串对象的引用。

程序中仅含有一个对象，即名为 country 的 String（字符型）对象（用更为专业的术语讲，country 是指向 String 对象的变量名）。

图 1.2 显示了 9 个变量和一个对象。每个变量均有一个名字和类型。当类型是一个类时，变量是指向那个类的实例（即一个对象）的一个引用，没有初始值的变量为空，其余的为初值。一个引用变量的惟一可能的值就是一个引用，用黑点表示。当引用不为空时，该点成为指向所引用的对象的箭头。在 Java 中，除非有某个引用变量引用某个对象，否则对象不能够存在。

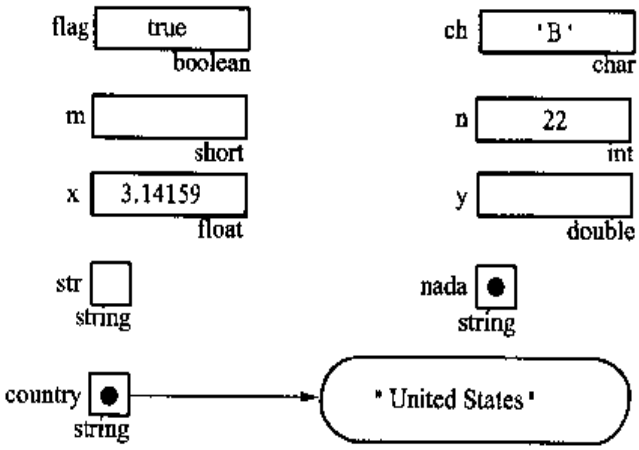


图 1.2　变量与对象

1.4　基本类型

Java 定义了 8 种基本类型：

boolean	布尔型，其值为 false 或 true。
char	16 位统一码，例如，'B' 或 'π'。
byte	8 位整数，取值范围为 -128～127。
short	16 位整数，取值范围为 -32 768～32 767。
int	32 位整数，取值范围为 -2 147 483 648～2 147 483 647。
long	64 位整数，取值范围为-9 223 372 036 854 775 808～9 223 372 036 854 775 807。
float	32 位浮点小数，取值范围为（正或负）1.4E-45F～3.4E38F。
double	64 位浮点小数，取值范围为（正或负）4.9E-324～1.8E308。

注意，浮点数在写法上用 F 作为后缀以便与类型 double 的值相区别。

每一个 Unicode 字符文字可以用'\uxxxx'的方式表示，其中 x 是任意的十六进制数字。例如，'B'的 Unicode 值为'\u0042'，而'π'为'\u03C0'。

除了数字值以外，浮点变量还可以有 3 个特殊的值——NEGATIVE_INFINITY、POSITIVE_INFINITY 和 NaN（表示非数字值）。这些特殊的值由非正常的算术操作所产生。

例 1.2　浮点类型中的特殊值

```
public class Ex0102
{ public static void main(String[] args)
  { double x=1E200;   //表示 1 后面跟 200 个 0
    System.out.println("x = " + x);
```

```
        System.out.println("x*x = " + x*x);
        System.out.println("(x*x)/x = " + (x*x)/x);
        System.out.println("(x*x)/(x*x) = " + (x*x)/(x*x));
    }
}
```

经编译后运行该程序，其输出将为：

```
x=1.0E200
x*x=Infinity
(x*x)/x=Infinity
(x*x)/(x*x)=NaN
```

在上例中，x*x 的值是 Infinity，这是由于 $10^{200}\times10^{200}=10^{400}$ 的原故，该值超出了 double 型值的最大值 1.8E308。

从代数角度上讲，(x*x) /x=x，但当 x 为无穷数时，情况并非如此，因为一个无穷数除以一个非负的有限数时仍然是无穷数，这就是 (x*x) /x=Infinity 的原因（此时，x*x 是一个无穷数，而 x 是一个有穷数）。

最后，一个无穷数除以一个无穷数甚至不是一个无穷数，这点从代数上讲是不确定的，因此，在 Java 语言中，其值为 NaN。

对 8 种基本类型中的每一种而言，Java 均提供了一个包装类（wrapper class），它为基本类型的操作提供面向对象的服务机制，例如，对 double 类型的包装类是 Double，类型 int 的包装类是 Integer。

例 1.3 包装类的使用

```
public class Ex0103
{ public static void main(String[] args)
  { String s = "2.7182818284590";
    System.out.println("s = " + s);
    Double x = new Double(s);
    System.out.println("x = " + x);
    double y = x.doubleValue();
    System.out.println("y = " + y);
    s += 45;
    System.out.println("s = " + s);
    y += 45;
    System.out.println("y = " + y);
    int n = x.intValue();
    System.out.println("n = " + n);
    n = Integer.parseInt("3A9",16);
    System.out.println("n = " + n);
  }
}
```

标准分享网 www.bzfxw.com 免费下载

经编译后运行该程序，其输出将为：

```
s=2.7182818284590
x=2.718281828459
y=2.718281828459
s=2.718281828459045
y=47.718281828459
n=2
n=937
```

在上述程序中，构造器 Double(s)将从 String 对象 s 生成 Double 对象 x，将数值 2.718 281 828 459 0 放入对象 x 中。第二个 println()语句将隐含地调用 Double.toString()方法，它将数值转换为字符串用于打印输出。而 x.doubleValue()调用将返回所存储的数值，并把该值赋给 y。+=操作将应用到 String 对象 s 上，对 double 类型变量 y 也进行看似同样的操作，但实际产生的结果是截然不同的。对字符串而言，"+="代表的操作语义是“添加”，即把第二个字符串添加到第一个字符串上，而对数值而言，"+="代表“增加”，即把第二个数加到第一个上面。

接下来，Double.intValue()方法将对存储的数值进行截取，产生整数 2，并用该值对 n 进行初始化。最后，Integer.parseInt("3A9",16)将对十六进制的数值 3A9 进行转换，返回十进制的整数值。因为 $3(16^2)+10(16)+9=937$，所以返回值为 937。

应注意到对字符串和数字值 y 进行+操作时，将自动地把数值 y 转换为该数值的字符串表示，如系统调用 System.out.println("y="+ y)中的+操作就是如此。

1.5 流程控制

Java 支持 if 语句、if…else 语句、switch 语句以及条件表达式运算符"?:"。

例 1.4 条件语句的使用

下面的程序将首先产生 0 到 99 间的随机整数，然后，利用条件语句来对他们进行分类。

```
public class Ex0104
{ public static void main(String[] args)
  { int n = (int)Math.round(100*Math.random());
    System.out.println("n = " + n);
    if (n>25 && n<75) System.out.println(n + " is between 25 and 75");
    else System.out.println(n + " is not between 25 and 75");
    switch ((int)n/20)
    { case  0: System.out.println(n + " < 20");
      case  1: System.out.println(n + " < 40");
      case  2: System.out.println(n + " < 60");  break;
      case  3: System.out.println(n + " < 80");  break;
```

```
      default: System.out.println(n - " >= 80");
    }
    System.out.println(n + (n%2>0 ? " is odd" : " is even"));
  }
}
```

该程序的输出为：

```
n=19
19 is not between 25 and 75
19<20
19<40
19<60
19 is odd
```

注意，在case为0和case为1的两个条件语句中都省略了break语句，这样，程序将顺序执行，一直到case为2的条件语句为止。在结果中，我们可以看到这3个输出语句的输出结果。对语句n%2 > 0 ? " is odd" : " is even"而言，表达的语义是：当条件部分n%2 > 0为真时（也就是n不能为2所整除），该语句的返回结果为" is odd"，否则返回结果为" is even"。

Java支持while语句、do..while语句以及for循环语句。

例1.5 循环语句的使用

下面的程序将从经验和尝试的角度来验证高斯（Guass）的素数定理，该定理可表述为：如p(n)表示小于n的素数的个数，则比值p(n)(ln n)/n随n的增大而趋向于1.0。

在程序中，奇数n的取值范围从3到1 000 000，对每个n，将计算出p(n)。随着p(n)的增加，我们在它达到5 000的倍数时进行输出，打印出相应的n、p(n)、ln n(自然对数)以及比值p(n)(ln n)/n，可以看到，它趋向于1.0。

```
class Ex0105
{ public static void main(String[] args)
  { System.out.println("n\tp(n)\tln(n)\t\t\tp(n)*ln(n)/n");
    final String DASHES18="\t------------------";
    System.out.println("------\t-----" + DASHES18 + DASHES18);
    int p=1;  // p 为小于 n 的素数的个数
    for (int n=3; n<1000000; n += 2)
    { int d=3;
      while (d<=Math.sqrt(n) && n%d>0)
        d += 2;
      if (n%d==0) continue;
      ++p;
      if (p%5000>0) continue;
```

标准分享网 www.bzfxw.com 免费下载

```
        double ln=Math.log(n);
        System.out.println(n + "\t" + p + "\t" + ln + "\t" + p*ln/n);
      }
      System.out.println("------\t-----" + DASHES18 + DASHES18);
    }
}
```

```
n          p(n)       ln(n)                P(n)*ln(n)/n
-----      ----       ----------------     ----------------
48619      5000       10.79176967999097    1.109830486023054
104743     10000      11.559265009775785   1.1035835339617717
163847     15000      12.006688344529985   1.0991981859170432
224743     20000      12.322712806131365   1.0966048158235286
287137     25000      12.567714732761953   1.0942263390613152
350381     30000      12.766776412829921   1.093105198012728
414991     35000      12.936012112230687   1.091012633835611
479939     40000      13.08141429147497    1.0902564110418174
545749     45000      13.209914442069696   1.0892299388420983
611957     50000      13.324417297588104   1.0886726761511107
679279     55000      13.428787220525182   1.087304770394617
746777     60000      13.52352189210366    1.0865510232990836
814309     65000      13.610095179831822   1.0863888114819662
882389     70000      13.690388280841916   1.0860597533048737
951193     75000      13.765472265206315   1.085384795609801
-----      -----      ----------------     ----------------
```

在程序中，for 循环重复执行了 499 999 次，分别对应 n=3,5,7,9,…,999 999 中的每一个值。循环体对 n 进行测试，看能否被 d=3,5,7,9,…,$\sqrt{n}$ 等数所整除，如果能整除（即 n%d 的余数为零），则第一个 continue 语句将被执行并重复执行下一个 for 循环。当 n 不能为上述各数整除时，表明 n 是一个素数，这样计数器 p 加 1。

Java 支持子程序，这些子程序称为方法。

例 1.6 方法的使用

下面这个程序与例 1.5 的输出结果是一样的，但它使用了一个单独的方法来判断某个整数 n 是否为素数。

```
public class Ex0106
{ public static void main(String[] args)
  { System.out.println("n\tp(n)\tln(n)\t\t\tp(n)*ln(n)/n");
    final String DASHES18="\t----------------";
    System.out.println("------\t-----" + DASHES18 + DASHES18);
    int p=1;  // p 为小于 n 的素数的个数
    for (int n=3; n<1000000; n += 2)
      if (isPrime(n))
```

```
      { ++p;
        if (p%5000>0) continue;
        double ln=Math.log(n);
        System.out.println(n + "\t" + p + "\t" + ln + "\t" + p*ln/n);
      }
    System.out.println("------\t-----" + DASHES18 + DASHES18);
  }

  private static boolean isPrime(int n)
  { int d=3;
    while (d<=Math.sqrt(n) && n%d>0)
      d += 2;
    if (n%d==0) return false;
    return true;
  }
}
```

程序中的 isPrime()方法封装了判断 n 是否为素数的代码。当 n 能被某个整数所整除时，就返回 false，表明它不是素数。反之，找不到能整除 n 的整数时，返回 true，表明它是一个素数。

值得指出的是，该方法被声明为 private static，这是有特别的含义的，声明为 private 意味着它将不为该类之外的其他类所使用，而说明为 static 则是由于它将被同样是 static 的 main()方法所调用。

1.6 类

一个类是一个抽象数据类型的实现。对象是通过对类进行实例化所产生的，在类定义中指明了该类的对象所能保持的数据种类和对象所能完成的操作，这些描述统称为类的成员。

如图1.3 所示，一个类有 4 种成员，分别称为 field、method、inner class 和 interface。field 指存储数据的变量，method 则是完成操作的函数，在类中，可以定义只用于该类的所谓 inner class 和 interface。图 1.3 中还列出了 4 种常用的特定方法 constructor、accessor、mutator 和 utility。

一个 constructor 是用于建立该类对象的方法，这一过程也称为实例化，所产生的对象称为该类的实例，该方法的另一个作用就是对所创建的对象中的各个 field 进行初始化。

一个 accessor 方法是一个只读函数，它返回类中某个 field 的值，这样就避免外部方法对 field 的直接修改。在 Java 语言中，访问某个 field X 的存取方法常取名为 getX()，正因为如此，Java 中的存取方法也常称为“getty”方法。

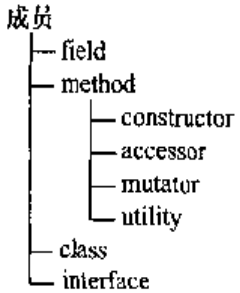


图 1.3 类的成员

标准分享网 www.bzfxw.com 免费下载

一个 mutator 方法是一个读写函数，允许外部方法调用该方法来改变类中某个 field 的值。在 Java 中，改变某个 field X 的 mutator 方法常取名为 setX()，正因为如此，Java 中的存取方法也常称为“setty”方法。

一个 utility 方法是一个私有函数，在该类中的其他方法内部使用。

例 1.7　类 Point

下面的程序中定义了一个 Point 类，它的实例可表达欧几里德平面中的点。

```
Public class point
{ protected double x,y;

  public Point(double x,double y)
   { this.x = x;
   this.y = y;
   }

   public double getX()
   { return x;
   }

   public double getY()
   { return y;
   }

   public Point getLocation()
   { return new Point(x,y);
   }

   public void setLocation(double x,double y)
   { this.x = x;
     this.y = y;
   }

public void translate(double dx, double dy)

 { x += dx;
   y += dy;
  }

public boolean equals( Object object)
  { if ( object == this) return true;
    if ( object.getClass() != this.getClass() ) return false;
    Point point= (Point)object;
```

```
    return( x = = point.x && y = = point.y );
  }

public int hashCode()
 { return (new Double(x_).hashCode() + (new Double(y) ).hashCode();
  }

public String toString()
{ return new String ("(" + (float)x + ", " + (float) y +" )" );
}

}
```

从上可以看出，该类有 x 和 y 两个 field 成员、一个构造函数和 getX()、getY()、getLocation()、hashCode()、toString()等 5 个存取方法以及一个 mutator 方法 translate(double,double)。

下面是对类 Point 进行操作测试的驱动程序：

```
public class Ex0107
{ public static void main(String[] args)
  { Point p = new Point(2,3);
    System.out.println("p = " + p);
    System.out.println("p.hashCode() = " + p.hashCode());
    Point q = p.getLocation();
    compare(p,q);
    q.translate(5,-1);
    compare(p,q);
    q = p;
    compare(p,q);
  }
  private static void compare(Point p, Point q)
  { System.out.println("q = " + q);
    System.out.println("q.hashCode() = " + q.hashCode());
    if (q.equals(p)) System.out.println("q equals p");
    else System.out.println("q does not equal p");
    if (q == p) System.out.println("q == p");
    else System.out.println("q != p");
  }
}
```

该测试程序的执行结果为：

```
p=(2.0,3.0)
p.hashCode()= -2146959360
q=(2.0, 3.0)
q.hashCode()= -2146959360
```

标准分享网 www.bzfxw.com 免费下载

```
q equals p
q != p
q=( 7.0, 2.0);
q.hashCode()= -2145648640
q does not equal p
q !=p
q=( 2.0, 3.0)
q.hashCode()= -2146959360
q equals p
q = = p
```

在该测试驱动程序中，包括了一个 utility 方法 compare(Point, Point)来辅助测试操作。

从输出可以看出，p 的 hashCode()值为-2 146 959 360。该数值没有特别的含义，仅仅用做一个标识，它是从对象的 x 和 y 成员通过计算得到的，q 开始时作为 p 的一个拷贝，因而具有同样的 hashCode()值，此时 equals(Object)方法返回值为 true。但是，测试 q == p 的返回值却为 false，这是因为 p 和 q 的确是不同的对象。相等运算符符 ==实际是检查是否相同的标识，而不是判断对象是否具有相同的值，只有在指向同一对象的两个不同引用变量间进行比较时才返回 true。

在 q 转换成 location(7,2)后，方法 equals(Object)返回值为 false。

最后，当引用 p 被赋值为 q 时，q == p 返回值为 true，这是因为此时仅有一个 Point 对象存在，也就是最初的 Point(2,3)对象，另一个 Point(7,2)对象将被回收，也就是说，当引用变量 q 失去了指向对象 Point(7,2)的引用后，该对象也就不存在了，参见图 1.4。

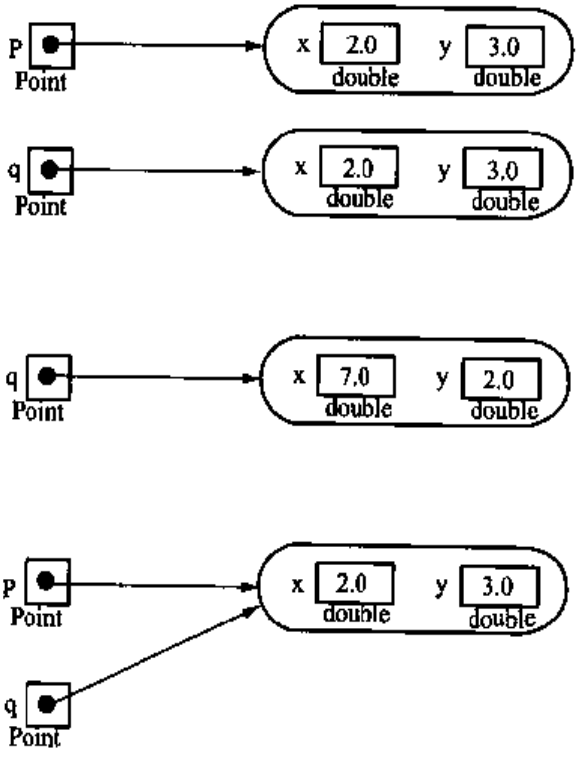


图 1.4 测试程序中各对象及其变化

1.7 修饰符

在对类、接口以及它们的成员进行声明时均可以带上修饰符，这些修饰符包括：public、protected、private、package、abstract、static 以及 final。表 1.1 对这 6 个修饰符的含义进行了归纳。

表 1.1 类、接口及其成员的修饰符

Modifier	Interf-ace	Class	Nested Class	Field	Method
public	可从任意类访问				
protected	只能在本类及其子类中访问				
private	只能在本类中访问				
abstract	N/A	至少包含一个 abstrxt 方法		N/A	其实现没有定义，声明的仅是其基调及返回类型
final	N/A	不能有子类		其值不能改变	不能被子类所覆盖
static	N/A	N/A	不是一个内部类	对类的所有对象而言仅有一个实例	对类的所有对象而言仅有一个实例

在上述 6 个修饰符中，public、protected 和 private 修饰符称做存取访问修饰符，因为这些修饰符将确定类或成员只能从哪些地方来进行访问。通常，public 意指可以用来自任何地方的方法访问，而 protected 指只能被本类或该类的子类的方法所访问，而 private 则只能由该类自身所访问。

当一个成员声明为 abstract 时，表明它是不完整的。因此一个 abstract 方法则意味着没有包括它的实现，而一个 abstract 类则至少有一个 abstract 方法。接口在缺省情况下为 abstract，所以也就不用声明为 abstract 了。值得指出的是，field 不能为 abstract。

一个 final 类是指不能有子类的类，而一个 final 型的 field 则意味着它是一个常量。一个 final 型的方法表明该方法不能在子类中被覆盖（overridden）。

一个 static 型的 field 成员是属于类本身的，而不是为该类的每个实例对象产生一个单独的拷贝。同样，static 型的方法也是属于整个类而不是属于某个对象的。如类 X 具有 static field x 和 static 型方法 y()，则它们可以由 X.x 和 X.y()所存取，而独立于任何的对象实例（见 1.9 节）中要使用的 java.lang.Math 类就是说明 static 方法是如何被使用的好例子。

1.8 String 类

一个字符串对象包含一个通常用做文本处理的字符序列，在 Java 中提供了 String 类来支持对字符串的建立和处理，该类包含的方法个数高达 50 个，并且包含 10 余个的构造器，一些常用的方法罗列在下面的类定义中：

```
public final class String
{ public char     charAt();
  public boolean  endswith(string suffix)
  public boolean  equals(Object object);
  public int      indexOf(char ch);
  public int      indexOf(char ch, int start);
```

标准分享网 www.bzfxw.com 免费下载

```
    public int      indexOf(String str);
    public int      indexOf(String str, int start);
    public int      lastIndexOf(char ch);
    public int      lastIndexOf(char ch, int start);
    public int      lastIndexOf(String str);
    public int      lastIndexOf(String str, int start);
    public int      length();
    public String   replace(char ch, char ch2);
    public boolean  startsWith(String prefix);
    public boolean  startsWith(String prefix, int start);
    public          String();
    public          String( char[] chars);
    public          String( char[] chars, int start, int len);
    public String   substring( int start);
    public String   substring(int start, int stop);
    public char[]   toCharArray();
    public String   toLowerCase();
    public String   toUpperCase();
    public String   trim();
}
```

尽管串是对象而不是基本类型，但它们在某些方面却与基本类型相似。系统对字符串文字（如“blue”和“'Tis so.”）的识别和处理与对数字文字（如 8388608 和 3.14159）的识别和处理是一样的。然而，串文字可以赋值给串引用，如：

```
String color = "blue";
```

因为文字是惟一的，任何指向"blue"的引用与 color 引用进行相同测试时，将返回真值。

正如数字类型一样，串是可通过运算符进行处理的惟一的对象类，其他类对象只能由类中的方法进行操作处理（见例 1.8）。

除此之外，String 类还定义了两个并置运算符+和+=，这些将在下面的例子中加以说明。

例 1.8　测试 String 类

```
public class Ex0108
{ public static void main(String[] args)
  { String s="ABCDEFG";
    System.out.println("s = \"" + s + "\"");
    s = s + "HIJK";
    System.out.println("s = \"" + s + "\"");
    s += "LMNOP";
    System.out.println("s = \"" + s + "\"");
```

```
    System.out.println("s.length() = " + s.length());
    System.out.println("s.charAt(6) = " + s.charAt(6));
    System.out.println("s.indexOf('G') = " + s.indexOf('G'));
    System.out.println("s.indexOf('Z') = " + s.indexOf('Z'));
    System.out.println("s.indexOf('G',8) = " + s.indexOf('G',8));
    System.out.println("s.indexOf(\"GHIJ\") = " + s.indexOf("GHIJ"));
    if (s.startsWith("DE"))

      System.out.println("s.startsWith(\"DE\")");
    else System.out.println("s does not start with \"DE\"");
    if (s.startsWith("DE",3))
      System.out.println("s.startsWith(\"DE\",3)");
    else
      System.out.println("s does not start with \"DE\" after 3 chars");
    if (s.endsWith("IJK")) System.out.println("s.endsWith(\"IJK\")");
    else System.out.println("s does not end with \"IJK\"");
    if (s.endsWith("NOP")) System.out.println("s.endsWith(\"NOP\")");
    else System.out.println("s does not end with \"NOP\"");
    s += "DABBADABBADO";
    System.out.println("s = \"" + s + "\"");
    s = s.replace('B','T');
    System.out.println("s = \"" + s + "\"");
    s = s.substring(7,10);
    System.out.println("s = \"" + s + "\"");
    s = s.toLowerCase();
    System.out.println("s = \"" + s + "\"");
    s = "  W XY  Z   ";
    System.out.println("s = \"" + s + "\"");
    System.out.println("s.length() = " + s.length());
    s = s.trim();
    System.out.println("s = \"" + s + "\"");
    System.out.println("s.length() = " + s.length());
  }
}
```

该程序的输出为：

```
s = "ABCDEFG"
s = " ABCDEFGHIJK"
s = "ABCDEFGHIJKLMNOP"
s.length() = 16
s.charAt(6) = G
sGindexOf(     = 6
sZindexOf(     = -1
sGindexOf(       = -1
```

标准分享网 www.bzfxw.com 免费下载

```
s.indexOf("GHIJ") = 6
s does not start with "DE"
s.startsWith("DE",3)
s does not end with "IJK"
s.endWith("NOP")
s = "ABCDEFGHIJKLMNOPDABBADABBADO"
s = "ATCDEFGHIJKLMNOPDATTADATTADO"
s = "HIJ"
s = "hij"
s = "   W   XY  Z   "
s.length() = 12
s = "W XY Z"
s.length() = 7
```

注意，String 对象是不能改变的，所以改变一个字符串的惟一办法是给它赋予一个方法的返回值，例如方法 replace (char,char)。

Java 需要计算机操作系统的支持才能维持一个字符串文字池，使得在任何运行环境下仅有一个相同的字符串文字在使用。这类似于数字，只能有一个 27。这个非同寻常的要求是为了保证对字符串文字实施正确的相等运算符 ==所提出的。

例 1.9 测试 String 文字的惟一性

```
public class Ex0109
{ public static void main(String[] args)
  { String s1="ABCDEFG";
    System.out.println("s1 = \"" + s1 + "\"");
    System.out.println("(s1 == \"ABCDEFG\") = " + (s1 == "ABCDEFG"));
    System.out.println("(s1 == \"ABCD\"+\"EFG\") = "
                    + (s1 == "ABCD"+"EFG"));
    String s2="ABCDEFG";  // makes s2 a synonym for s1
    System.out.println("s2 = \"" + s2 + "\"");
    System.out.println("(s1 == s2) = " + (s1 == s2));
    s2 = new String("ABCDEFG");  // now s2 refers to a separate object
    System.out.println("s2 = \"" + s2 + "\"");
    System.out.println("(s1 == s2) = " + (s1 == s2));
    System.out.println("s1.equals(s2) = " + s1.equals(s2));
  }
}
```

该程序的结果为：

```
s1="ABCDEFG"
(s1 == "ABCDEFG") =true
(s1 =="ABCD"+"EFG") =true
s2="ABCDEFG"
```

```
(s1 ==s2) = true
s2="ABCDEFG"
(s1 ==s2) =false
s1.equals(s2)=true
```

上例说明仅有一个字符串文字"ABCDEFG"，因此，当两个不同的引用（即引用具有不同的名字）均被赋值成该文字时，它们必须相同。也就是说，相等运算符==应为 true。但请注意两个不同的字符串可以有相同的文字值，在这种情况下，相等运算符的返回值为假，而 equals()方法仍然为真。

这些特性使得 String 类与其他类不同：

- String 对象是不可改变对象，这意味着它们的值是不能改变的，它们是只读对象，因为没有为该类定义 mutator 方法。
- String 文字均由操作系统的字符串池所维持。
- String 类中定义了两个特定的运算符+和+=。
- String 类对象的字符长度由 length()方法获得，这与数组对象所用的 length 成员或对集合对象所采用的 size()方法不同。

1.9 Math 类

Java 语言中的 Math 类定义了数学常量和一些常用的数学函数，它们位于 java.util 包中，其定义如下所示：

```
public final Math
{ public static final double  E=2.7182818284590452354;
  public static final double  PI=3.14159265358979323846;
  public static double        abs(double x);      // 绝对值
  public static native double atan(double x);    // 反正切值
  public static native double ceil(double x);    // 天花板值
  public static native double cos(double x);
  public static native double exp(double x);     // e为底的指数值
  public static native double floor(double x);
  public static native double log(double x);     // e为底的对数
  public static native double max(double x, double y);
  public static native double min(double x, double y);
  public static native double pow(double x, double y);    // 幂指数
  public static synchronized double random ();
  public static long          round(double x);
  public static native double  sin(double x);
  public static native double  sqrt(double x);    // 平方根
  public static native double  tan(double x);
}
```

标准分享网 www.bzfxw.com 免费下载

请注意，这些成员均是 static 的，也就意味着可以用"Math."前缀的方式来对它们进行调用，而不要使用 Math 类的某个对象。

例 1.10 测试 Math 类

```
public class Ex0110
{ public static void main(String[] args)
  { final double PI=Math.PI;
    final double E=Math.E;
    System.out.println("E                    = " + E);
    System.out.println("Math.exp(1.0)        = " + Math.exp(1.0));
    System.out.println("PI                   = " + PI);
    System.out.println("4*Math.atan(1.0)     = " + 4*Math.atan(1.0));
    System.out.println("Math.cos(2*PI)       = " + Math.cos(2*PI));
    System.out.println("Math.sin(PI/2)       = " + Math.sin(PI/2));
    System.out.println("Math.tan(PI/4)       = " + Math.tan(PI/4));
    System.out.println("Math.log(E)          = " + Math.log(E));
    System.out.println("Math.abs(-13.579)    = " + Math.abs(-13.579));
    System.out.println("Math.floor(13.579)   = " + Math.floor(13.579));
    System.out.println("Math.ceil(13.579)    = " + Math.ceil(13.579));
    System.out.println("Math.round(13.579)   = " + Math.round(13.579));
    System.out.println("Math.pow(25.0,0.5)   = " + Math.pow(25.0,0.5));
    System.out.println("Math.sqrt(25.0)      = " + Math.sqrt(25.0));
    System.out.println("Math.random()        = " + Math.random());
    System.out.println("Math.random()        = " + Math.random());
  }
}
```

该程序的输出为：

```
E                    = 2.718281828459045
Math.exp(1.0)        = 2.7182818284590455
PI                   = 3.141592653589793
4*Math.atan(1.0)     = 3.141592653589793
Math.cos(PI/2)       = 1.0
Math.sin(PI/2)       = 1.0
Math.tan(PI/4)       = 0.9999999999999999
Math.log(E)          = 1.0
Math.abs(-13.579)    = 13.579
Math.floor(13.579)   = 13.0
Math.ceil(13.579)    = 14.0
Math.round(13.579)   = 14
Math.pow(25.0,0.5)   = 5.0
Math.sqrt(25.0)      = 5.0
Math.random()        = 0.9279776738566742
Math.random()        = 0.4493770111566855
```

注意，Math.tan(PI/4)方法在计算时的细微差错，其正确的值应当为 1.0。同时，应注意到 Math.round(double) 返回的值类型为 long 型整数而不是 double 型十进制浮点数，而 Math 类中其他方法的返回结果均是 double 型的。

Math.random()方法随机产生一个在 0.0 到 1.0 之间均匀分布的双精度十进制浮点数，下面的程序将对这种分布进行测试。

例 1.11 测试 Math.random()方法

下面的程序产生 0.0 到 1.0 间的 50 000 个随机数，对区间 0.0 到 1.0 进行 100 等分，子区间的宽度为 0.01,然后对落在各个区间的随机数个数进行统计，该程序使用 frequency[]数组来记录累加结果。例如，随机数 x=0.7294416115902632 落在区间 0.72 到 0.73 中，所以使 frequency[72]加 1。

```
public class Ex0111
{ public static void main( String[ ] args )
  { final int SUBINTERVALS = 100;
    final int TOTAL = 50000;
    int [ ] frequency = new int[SUBINTERVALS];
    for ( int k=0; k< SUBINTERVALS; k++)
    frequency[k]=0;
  for(int j=0; j<TOTAL; j++ )
   {
     double x=Math.random();          // 0.0 < x < 1.0
     x *= SUBINTERVALS;               // 0.0 < x < 100.0
     x = Math.floor(x);               // 0.0 < = x < =99.0
     long m = Math.round(x);          // 0 < = m < = 99
     int k = (int)m;                  // 0 < = k < = 99
     ++frequency[k];                  // count  k,对所在区间的数的个数进行计数
      }
   for( int i=0; i< SUBINTERVALS;i++)
     System.out.print(frequency[i]+(i%10==9?"\n":"  "));
   }
}
```

该程序的输出结果为：

```
515  506  515  454  495  512  502  542  506  514
475  529  483  504  476  496  471  524  508  489
501  499  477  498  500  498  517  482  438  492
491  480  566  487  512  485  489  461  524  504
500  527  482  506  498  475  519  517  466  509
473  503  497  483  522  468  505  510  488  532
499  533  499  488  511  468  506  476  510  498
552  504  507  478  512  521  484  479  480  487
```

标准分享网 www.bzfxw.com 免费下载

```
514   489   511   512   464   487   499   506   497   526
511   501   522   468   524   505   508   533   530   504
```

假如上述 50 000 个随机数均匀地落在各个子区间中的话，则每个子区间中应有 500 个。从上述程序的结果来看几乎是接近 500 的，这也就从实验的角度得到了验证。因此有理由相信 Math.random()方法确实能够产生分布均匀的随机数。

例 1.12 测试 Math.sqrt()方法

下面的程序将对 Math.sqrt()方法进行测试。在程序中，定义了一个局部求解平方根的方法，我们将 Math.sqrt()所返回的值及该值的平方与该局部方法所获得的值进行比较：

```
public class Ex0112
{ public static void main(String[] args)
  { for (double x=50.0; x<60.0; x++)
    { double y=Math.sqrt(x);
      double z=sqrt(x);
      System.out.println(y + "\t" + y*y);
      System.out.println(z);
    }
  }
  private static double sqrt(double x)
  { final double EPSILON=1E-14;
    if (x <= 0) return 0.0;
    double y=1.0;
    while (Math.abs(y*y-x) > EPSILON)
      y = (y+x/y)/2;
    return y;
  }
}
```

该程序的输出结果为：

```
7.0710678118654755              50.00000000000001
7.0710678118654755
7.141428428542 85               51.00000000000001
7.141428428542 85
7.211102550927978               51.99999999999999
7.211102550927979
7.280109889280518               53.0
7.280109889280518
7.3484692283495345              54.0
7.3484692283495345
7.416198487095663               55.0
7.416198487095663
7.483314773547883               56.0
```

```
7.483314773547883
7.54983443527075                    57
7.549834435270 75
7.615773105863909                   58.00000000000001
7.615773105863909
7.681145747868608                   58.99999999999999
7.681145747868609
```

复 习 题

1.1 为什么说 Java 是一种面向对象的程序设计语言?
1.2 对一个 Java 程序来说什么是必需的?
1.3 基本类型是什么?
1.4 什么是文字?
1.5 对两个浮点数类型而言，3 个非数字文字是什么?
1.6 什么是对象引用?
1.7 什么是对象?
1.8 什么是类?
1.9 什么是实例?
1.10 什么是 field 成员?
1.11 方法是什么?
1.12 方法的说明指什么?
1.13 什么是构造函数?
1.14 什么是静态成员?
1.15 存取描述符有哪些?
1.16 存取描述符中的 private 是什么含义?
1.17 存取描述符中的 protected 是什么含义?
1.18 什么是包装类（wrapper class）?
1.19 是什么使得 String 类与 Java 标准类库中的其他类如此不同?
1.20 Math.random()方法的作用是什么?

习 题

1.1 编写如下方法并对其进行测试：

```
public static String monthName(int month)
// 前提条件： 0<n<13
// 结果示例：monthName(2)返回"February"
```

标准分享网 www.bzfxw.com 免费下载

1.2 编写并测试如下方法：

```
public static int daysInMonth(inL month, int year)
// 前提条件： 0<n<13
// 结果示例：daysInMonth(2, 2000)返回 29
```

1.3 编写并测试如下方法：

```
public static int numberOfDigits(int n)
//  结果示例：numberOfDigits(-8036800)返回 7。
```

1.4 编写并测试如下方法：

```
public static int sumOfDigits(int n)
// 结果示例：sumOfDigits (-8036800)返回 25。
```

1.5 编写并测试如下方法：

```
public static int reverseDigits(int n)
// 结果示例：reverseDigits(-8036800)返回 572038
```

1.6 编写并测试如下方法：

```
public static double round(double x, int precision)
// 结果示例：round(803.505692, 4) 返回 803.5057
```

1.7 编写并测试如下方法：

```
public static String signedToBinary(int n)
// 结果示例：signedToBinary(1289)返回"010100001001"
//          signedToBinary(-1289)返回"101011110111"
```

1.8 编写并测试如下方法：

```
public static String unsignedToBinary(int n)
// 前提条件：n>=0
// 结果示例：unsignedToBinary(1289)返回"010100001001"
```

1.9 编写并测试如下方法：

```
public static int binaryToSigned(String code)
// 前提条件：Code 中的每个字符要么是'0'，要么是'1'；
            Code.length <=32
// 结果示例：binaryToSigned("010100001001")返回 1289
//          binaryToSigned("101011110111")返回 -1289
```

1.10 编写并测试如下方法：

```
public static int binaryToUnsigned(String code)
// 前提条件：Code中的每个字符要么是'0'，要么是'1'：
            Code.length <=32
// 结果示例：binaryToUnsigned("010100001001")返回1289
//          binaryToUnsigned("101011110111")返回2807
```

1.11 编写并测试如下方法：

```
public static String format(String s, int len, int d)
// 结果示例：format("tomato",9,d)，当d为-1,0,1时将分别返回
//          "tomato   ", " tomato ", "  tomato"
```

1.12 编写并测试如下方法：

```
public static int randomInt(int start, int stop)
// 前提条件：start < stop ;
// 返回位于[start,stop-1]区间中均匀分布的整型数。
```

1.13 实现Point类的缺省构造函数（见例子1.7）：

```
public Point()
// 构造一个表示原点（0，0）的Point对象。
```

1.14 在Point类中增加一个public static final型对象以表示原点（0，0）。

1.15 为Point类实现以下拷贝构造函数（见例子1.7）：

```
public Point(Point point)
// 构造一个Point对象，该对象的坐标值与给定的Point对象一样。
```

1.16 为Point类实现以下方法（见例子1.7）：

```
public double distance(Point point)
// 返回该对象与指定Point对象间的欧几里德距离。
```

1.17 为Point类实现以下方法（见例子1.7）：

```
public double magnitude()
// 返回该对象与原点间的欧几里德距离。
```

1.18 为Point类实现以下方法（见例子1.7）：

```
public double amplitude()
// 返回该点的极坐标角（弧度）。
```

标准分享网 www.bzfxw.com 免费下载

1.19 为 Point 类实现以下构造函数（见例子 1.7）：

```
public void setPolar(double r, double theta)
// 将该点移到到指定的极坐标 (r,theta)。
```

1.20 为 Point 类实现以下方法（见例子 1.7）：

```
public static Point polar(double r, double theta)
// 返回极坐标为 (r,theta) 的点对象。
```

1.21 为 Point 类实现以下方法（见例子 1.7）：

```
public void expand(double dr )
// 将该点放大 dr 倍。
```

1.22 为 Point 类实现以下方法（见例子 1.7）：

```
public void rotate(double theta )
// 将该点按顺时针方向旋转 theta 弧度。
```

复习题答案

1.1 Java 被称为面向对象的程序设计语言是因为所有的数据和操作均封装在对象中，而且对象的类均在一个单继承的层次结构中定义。

1.2 一个 Java 程序必须有一个 public 类，该类定义在名为 X.java 的文件中，这里的 X 是类的名字，类必须包括一个如下所示的方法：

```
public static void main( String[ ] args)
```

1.3 一个基本类型是指以下类型之一：boolean、char、byte、short、int、long、float、double。

1.4 文字是一个没有名字的常量，是表示该类型值的一个符号。例如，44、3.14 以及“East Anglia”均为文字。

1.5 浮点数类型的 3 个非数字文字是 NEGATIVE_INFINITY、POSITIVE_INFINITY 以及 NaN。

1.6 对象引用是一个变量，其值要么为空，要么是某个对象的地址。

1.7 对象在内存中是一块连续的存储区，其类型由类所确定并通过该类的一个引用变量进行存取。对象可以包含保存数据的数据字段和完成操作的方法。

1.8 类是创建对象的蓝图，类定义中说明了类实例对象的数据字段和方法。

1.9 一个类的实例是该类所说明的类型的一个对象，一个实例的创建也称为类的实例化。

1.10 一个 field 是类的一个成员变量，当类实例化以后，结果对象的 field 用于保存对

象的数据。

1.11 方法是类的成员函数，当类实例化后，结果对象的方法将为对象完成各种操作。

1.12 方法说明是方法定义的一部分，编译器需要这些信息来对调用该方法的语句进行编译处理。例如，主方法的说明为main(String[])。

1.13 构造函数是类的一个方法，对类进行实例化。它的名字与类相同并且没有返回类型，它用new运算符进行调用。

1.14 一个类的静态成员是应用于整个类而不是类的某个特定对象的成员，它用类名而不是某个对象名进行调用。例如，类Math中的方法sqrt(double)就是一个静态方法，它以Math.sqrt()的形式进行调用。一个静态字段仅有一个数据值而不管有多少个对象（或没有实例对象）。

1.15 存取描述符指Jvaa中的以下3个保留字：public、protected或private。它们对类、字段、方法、接口等进行访问限制说明。

1.16 存取描述符中的private指该修饰实体只能在所在类的范围内进行访问。

1.17 存取描述符中的protected指该修饰实体只能在被定义的类的范围或该类的子类中进行访问。

1.18 包装类（wrapper class）也是一个类，其目的是为处理8个基本类型中的某个数据类型提供常量和方法。例如，Double类就是为double基本类型而定义的包装类。

1.19 String类与Java标准类库中的其他类不同，因为它的实例是不能改变的（只能读），它的文字具有惟一性并有两个进行串接操作的运算符+和+=。

1.20 Math.random()方法返回一个在0.0~1.0之间进行均匀分布的随机双精度浮点数。

习题解答

1.1
```
public static String monthName(int month)
{ switch (month)
  { case  1: return "January";
    case  2: return "February";
    case  3: return "March";
    case  4: return "April";
    case  5: return "May";
    case  6: return "June";
    case  7: return "July";
    case  8: return "August";
    case  9: return "September";
    case 10: return "October";
    case 11: return "November";
    case 12: return "December";
  }
  return "";
```

标准分享网 www.bzfxw.com 免费下载

```
    }
  }
```

1.2
```
public static int daysInMonth(int month, int year)
{ if (month==4 || month==6 || month==9 || month==11) return 30;
  if (month==2)
    if (year%400==0 || year%100!=0 && year%4==0) return 29;
    else return 28;
  return 31;
}
  }
```

1.3
```
public static int numberOfDigits(int n)
{ if (n<0) n = -n;
  int count=0;
  while (n>0)
  { n /= 10;
    ++count;
  }
  return count;
}
```

1.4
```
public static int sumOfDigits(int n)
{ if (n<0) n = -n;
  int sum=0;
  while (n>0)
  { sum += n%10;
    n /= 10;
  }
  return sum;
}
```

1.5
```
public static int reverseDigits(int n)
{ if (n==0) return 0;
  int sign = (n<0?-1:1);
  if (n<0) n = -n;
  int reverse=0;
  while (n>0)
  { reverse = 10*reverse + n%10;
    n /= 10;
  }
  return sign*reverse;
}
```

1.6
```
public static double round(double x, int precision)
{ double pow10 = Math.pow(10,precision);
  return Math.round(x*pow10)/pow10;
```

```
}
```

1.7
```
public static String signedToBinary(int n)
{ if (n==0) return "0";
  if (n>0) return "0" + unsignedToBinary(n); // Problem 1.8
  int mod=1;
  while(mod+2*n<0)
    mod *= 2;
  return unsignedToBinary(mod+n);
}
```

1.8 编写并测试unsignedToBinary(int n)方法。

```
public static String unsignedToBinary(int n)
{ // PRECONDITION: n > 0
  String code="";
  while (n > 0)
  { code = "" + (n%2) + code;  // append next bit to left of code
    n /= 2;  // remove current least significan bit
  }
  return code ;
}
```

1.9
```
public static int binaryToSigned(String code)
{ int len = code.length();
  int unsigned = binaryToUnsigned(code);  // Problem 1.10
  if (code.charAt(0) == '0') return unsigned;
  return  unsigned - (int)Math.pow(2,len);
}
```

1.10
```
public static int binaryToUnsigned(String code)
{ int n = code.length();
  int answer = 0;
  for (int i=0; i<n; i++)
    answer = answer*2 + (code.charAt(i)=='1' ? 1 : 0);
  return answer;
}
```

1.11
```
public static String format(String s, int len, int d)
{ int spaces = (len - s.length());
  if (spaces <= 0) return s;
  String formatS = "";
  int leftSpaces = (d<0)? 0 : ((d>0)? spaces : spaces/2);
  int rightSpaces = spaces - leftSpaces;
  for (int i=0; i<leftSpaces; i++)
    formatS += " ";
  formatS += s ;  // s comes after spaces on left
```

标准分享网 www.bzfxw.com 免费下载

```
    for (int i=0; i<rightSpaces; i++)
      formatS -= " ";
    return formatS;
  }
```

1.12
```
static java.util.Random random = new java.util.Random();
public static int randomInt(int start, int stop)
{ return start + random.nextInt(stop-start);
}
```

1.13 实现 Point 类的缺省构造函数。

```
public Point()
{ this.x = 0;
  this.y = 0;
}
```

1.14 在 Point 类中增加一个 public static final 型对象，以表示原点（0，0）。

```
public static final Point ORIGIN = new Point();
```

1.15 为 Point 类实现拷贝构造函数。

```
public Point(Point q)
{this.x = q.x;
 this.y = q.y;
}
```

1.16 为 Point 类实现 distance(Point point)方法。

```
public double distance(Point point)
{ double dx = this.x - point.x;
  double dy = this.y - point.y;
  return Math.sqrt(dx*dx+dy*dy);
}
```

1.17 为 Point 类实现 magnitude()方法。

```
public double magnitude()
   { return distance(ORIGIN);
   }
```

1.18 为 Point 类实现 amplitude()方法。

```
public double amplitude()
 { return Math.atan(y/x);
  }
```

1.19 为 Polar 实现 Point 类方法。

```
public void setPolar(double r, double theta)
```

```
{ this.x = r*Math.cos(theta);
  this.y = r*Math.sin(theta);
}
```

1.20 为Polar实现另一个point类方法。

```
public static Point polar(double r, double theta)
 { double x = r*Math.cos(theta);
   double y = r*Math.sin(theta);
   return new Point(x,y);
   }
```

1.21 为Point类实现expand(double dr)方法。

```
public void expand(double dr)
  { x *= dr;
    y *= dr;
  }
```

1.22 为Point类实现rotate(double theta)方法。

```
public void rotate(double theta)
{ double xx = x;
  double yy = y;
  double sin = Math.sin(theta);
  double cos = Math.cos(theta);
  x = xx*cos - yy*sin;
  y = xx*sin + yy*cos;
}
```

标准分享网 www.bzfxw.com 免费下载

第2章 数组回顾

一个数组是一个对象，它由同类型的一组成份元素所组成，每个元素均有一个惟一编号。元素的编号从0开始，因而对每个元素的引用可以由编号和下标运算符[]来进行，由于数组的高效性，它得到了非常广泛的使用。

2.1 数组的特性

Java中数组具有以下主要特性：

1. 数组是对象。
2. 数组是在运行时动态生成的。
3. 可以把数组赋给类型为Object的变量。
4. 在数组上可以调用适用于Object的任何方法。
5. 一个数组对象包含一组具有相同数据类型的变量。
6. 数组中的变量也称为数组的成份（components）。
7. 如果数组的成份类型是*T*，则数组的类型为*T*[]。
8. 一个数组类型变量将拥有指向一个数组对象的引用。
9. 数组的成份类型本身可以是数组类型。
10. 数组中不为数组类型的成份称为基元（element）。
11. 基元的类型可以是基本类型或引用类型。
12. 数组的长度是数组成份的个数。
13. 数组的长度在数组生成时设置，并不能随后改变。
14. 数组的长度可以public final型实例变量的形式存取。
15. 数组成份元素用0到length-1间的整形值来索引。
16. 当违背特性15时，将产生ArrayIndexOutofBoundsException异常。
17. 类型为short、byte和char的变量可用作数组索引。
18. 数组可以调用Object.clone()方法进行复制。
19. 数组可以调用Arrays.equals()进行相等性测试。
20. 每个数组对象均实现了Cloneable和java.io.Serializable接口。

可以看到，特性3可从特性1得到。尽管数组类型不是class类型，但它们以Object类的扩展形式的面貌出现。特性7表明数组类型与class类型是不一致的，事实上，数组类型是一种导出（derived types）类型，对每种类型*T*，存在一个相应的数组类型*T*[]。

同样，对8种基本类型的每一种，都存在相应的数组类型。

特性 9 允许数组的数组存在。从技术的角度上讲，Java 仅能有成份为基本类型的多维数组，但从对象的角度上说，数组的数组与数组是一回事。这是因为数组本身就是对象，数组的数组就成为对象数组，这其中的某个成份也可以不是数组，只要是对象就行（例 2.1 将对此进行说明）。

应当注意到，特性 13 所导致的结果是：当把一个引用成份的值变为 null 时，对数组的长度将不产生任何影响，null 仍是引用成份的一个有效值。

例 2.1 数值定义

下面是一些正确的数组定义例子：

```
public class Ex0201
   { public static void main(String[] args)
  { float x[];
    x = new float[100];
    args = new String[10];
    boolean[] isPrime = new boolean[1000];
    int fib[] = { 0, 1, 1, 2, 3, 5, 8, 13 };
    short[][][] b = new short[3][8][5];
    double a[][] = { {1.1,2.2}, {3.3,4.4}, null, {5.5,6.6}, null };
    a[4] = new double[66];
    a[4][65] = 3.14;
    Object[] objects = { x, args, isPrime, fib, b, a };
  }
}
```

程序的第一行声明 x[]为一浮点数组，但并没有分配任何存储空间。第二行对 x[]进行定义，并使之具有100个数组成份。

第三行声明 args[]为 String 对象数组。但请注意声明数组的两种等效的方法，方括号后缀可放在类型标识符或数组变量上，第四行使得 args[]具有10个类型为 String 的成份。

第五行定义了具有1 000个布尔型变量的数组 isPrime[]。

第六行定义了 8 个整数的数组 fib[]，并对其进行初始化，例如，fib[4]的值为3，fib[7]的值为13。

第八行定义数组 a[][]为有 5 个成份的数组，而每个成份本身又是数组，数组成份为 double 类型，5 个成份数组中仅有 3 个被分配，随后的代码行为 a[4]分配一个有 66 个成份的 double 类型数组，并对最后一个成份进行赋值，其值为 3.14。

最后一行定义了一个具有 6 个成份元素，其类型为 Object 的数组，每个成份又是数组，前 4 个成份数组中的成份类型均为基元（非数组），而 b 和 a 中的成份就不是基元，因为它们是数组类型。objects 中的基元包括 2，5，13（成份 fib 中的成份），空 null（成

标准分享网 www.bzfxw.com 免费下载

份 a 中的成份），2.2 以及 3.14（成份 a 的成份的成份）。

例 2.1 中的 a[][]称为粗糙数组（ragged array），因为它是一个具有不同长度的成份的二维数组。

在 Java 中，数组的成份类型可以是基本数据类型，引用类型或数组类型，当然，最简单的还是成份为基本数据类型的数组，如例 2.1 中的 x[]、isPrime[]以及 fib[]等，这些数组均可以进行排序。

2.2 数组复制

数组是一个对象，所以可以调用 Objext.clone（）方法来对它进行复制，这一点可从例2.2 中看出。

例 2.2 复制一个数组

```
public class Ex0202
{ public static void main(String[] args)
  { int[] a = { 22, 44, 66, 88 };
    print(a);
    int[] b = (int[])a.clone( );  // duplicate a[] in b[]
    print(b);
    String[] c = { "AB", "CD", "EF" };
    print(c);
    String[] d = (String[])c.clone( );  // duplicate c[] in d[]
    print(d);
    c[1] = "XYZ";  // change c[], but not d[]
    print(c);
    print(d);
  }
  public static void print(int[] a)
  { for (int i=0; i<a.length; i++)
      System.out.print(a[i] + " ");
    System.out.println( );
  }
  public static void print(Object[] a)
  { for (int i=0; i<a.length; i++)
      System.out.print(a[i] + " ");
    System.out.println( );
  }
}
```

该程序的输出结果为：

```
22 44 66 88
```

```
22 44 66 88
AB CD EF
AB CD EF
AB XYZ EF
AB CD EF
```

在程序中，数组a[]包含4个整型基元，而数组b[]是数组a[]的复制品，同样，数组d[]是数组c[]的复制品，每个数组均包含3个类型为String的成份。在这两种情况下，复制均是由对方法clone()的调用来完成的。由于方法clone()的返回类型为Object，它必须强行转换为复制的数组的类型int []或String []。

该程序的其余部分表明被复制的数组d[]确是数组c[]的一个单独备份，当把c[1]改变成"XYZ"时，d[1]中的值仍是"CD"。

2.3 Arrays类

在java.util.Arrays类中定义的方法有：

```
public static List asList(Object [ ] )
public static int binarySearch(…)
public static boolean equals(…)
public static void fill(…)
public static void sort(…)
```

这里，…表明该方法是覆盖的，对不同的参数类型有不同的方法实现，一种是对基本数据类型的，另一种针对引用类型。例如，binarySearch () 方法就有9个版本，其中7个是针对基本数据类型的（除boolean类型没有外），另外两个是针对类型为Object[]的数组。

应当注意到Arrays中的所有方法均是static型的，这样，它们可以用前缀为Arrays.的形式调用其中的方法，而不必用到类的实例名。事实上，对Arrays类进行实例化是不可能的，这是因为它的构造器被声明为private型。

例2.3 测试Arrays类中的方法

```
import java.util.Arrays;

public class Testing
{ public static void main(String[] args)
  { char[] a = new char[64];
    Arrays.fill(a,'H');
    String s = new String(a);
    System.out.println("s = \"" + s + "\"");
    Object[] objects = new Object[8];
    Arrays.fill(objects,2,5,"Java");
```

.标准分享网 www.bzfxw.com 免费下载

```
    System.out.println("objects = " + Arrays.asList(objects));
    int[] x = { 77, 44, 99, 88, 22, 33, 66, 55 };
    int[] y = (int[])x.clone( );
    System.out.print("x = "); print(x);
    System.out.print("y = "); print(y);
    System.out.println("Arrays.equals(x,y) = " + Arrays.equals(x,y));
    System.out.println("y.equals(x) = " + y.equals(x));
    y[4] = 0;
    System.out.print("y = "); print(y);
    System.out.println("Arrays.equals(x,y) = " + Arrays.equals(x,y));
    System.out.print("x = "); print(x);
    Arrays.sort(x);
    System.out.print("x = "); print(x);
    int i = Arrays.binarySearch(x,44);
    System.out.println("Arrays.binarySearch(x,44) = " + i);
    i = Arrays.binarySearch(x,47);
    System.out.println("Arrays.binarySearch(x,47) = " + i);
  }

  private static void print(int[] a)
  { System.out.print("{ " + a[0]);
    for (int i=1; i<a.length; i++)
      System.out.print(", " + a[i]);
    System.out.println(" }");
  }
}
```

该程序执行后结果为：

```
s="HHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHH"
objects=[ null, null, Java, Java, Java, null, null, null]
x={77, 44, 99, 88, 22, 33, 66, 55 }
y={77, 44, 99, 88, 22, 33, 66, 55 }
Arrays.equals(x,y)=true
y.equals(x,y)=false
y={77, 44, 99, 88, 0, 33, 66, 55 }
Arrays.equals(x,y)=false
x={77, 44, 99, 88, 22, 33, 66, 55 }
x={22, 33, 44, 55, 66, 77, 88, 99 }
Arrays.binarySearch(x,44)=2
Arrays.binarySearch(x,47)= - 4
```

程序中的前三行展示如何重复利用某个字符来构造一个 String 对象。起先，我们定义一个 64 个字符的数组，然后利用 Arrays.fill()方法来填充某个想要填充的字符，在本程序中，就是字符'H'，然后利用适当的构造器将该字符串构造成一个 String 型对象。

随后，我们构造一个名为 objects 的数组，该数组有 8 个引用成份。利用 String 型对象“Java”对该数组的成份 2 到 4 进行填充。请注意方法中的整型参数 2 和 5，它们用于指示填充的范围：其中开始的整数 2，是开始填充的数组下标，而第二个整数是随后开始的不被填充的数组成份的下标。这种标明范围的方法在标准库中得到广泛使用。该表示协议的结果是范围中的成份的个数等于这两参数之差。在本程序中，5–2 =3，因此，将有 3 个成份得到改变，它们是第 2、第 3 和第 4 个。

对数组对象而言，没有 toString()方法，所以不像其他的对象，它们不能直接打印输出。但是，方法 Arrays.toList()可产生一个 List 对象，而 List 对象具有 toString()方法，因而可以传递给 System.out.println()方法进行输出，这是打印一个数组对象的简单方式。

程序的接下部分将产生两个整型数组 x 和 y，使用 Object.clone()方法把 x 复制到 y。然后调用 Arrays.equals()方法以及 Object.equals()来对它们做相同检查。注意，在这两个方法调用中只有第一个返回真。然后改变 y[4]中的内容，并再次调用 Arrays.equals()方法，当两个数组内容不同时，该方法也能正常工作。

程序中的 Arrays.sort()方法是用于对 x 进行排序的，排序后，我们使用 Arrays.binarySearch()方法来对数组 x 中的特定整数进行查找，如该基元位于数组之中，查找方法将返回它在数组中的位置索引，对应于本程序而言，将返回 2。反之，但不存在于数组中时，将返回一个负数，以表明该成份不在本数组之中。对一般的折半查找将在 2.5 节中说明。

值得注意的是，Arrays.binarySearch()方法要求数组是事先排序的，否则将不能正常工作。

对数组而言，方法 Arrays.equals()是在代替 Object.equals()的意义上使用的，我们知道，对应方法 Object.equals()，只有当两个引用指向同一对象时才为真，也就是说，它检验两个是否为同一对象标识而不是检查它们的内容是否相同。

为了说明本书中所给出的有关数组的一些算法，我们定义另外的一个 Arrays 类，为避免名字冲突，把该类放到一个单独的名为 schaums.dswj 的包中，该名字正好代表 Schaum 的 Java 数据结构概览英文首字母缩写（Schaum's Outline of Data Structures with Java）。

例 2.4 一个 Arrays 实用类

```
package schaums.dswj;
import java.util.Random;

  public class Arrays
  { private static Random random = new Random( );

 public static int load(int start, int range)
 { return random.nextInt(range) + start;
 }

 public static void load(int[] a, int start, int range)
```

标准分享网 www.bzfxw.com 免费下载

```
{ int n=a.length;
  for (int i=0; i<n; i++)
    a[i] = random.nextInt(range) + start;
}

public static void print(int[] a)
{ for (int i=0; i<a.length; i++)
    System.out.print(" " + (i>9?"":" ") + i);
  System.out.print("\n{ " + a[0]);
  for (int i=1; i<a.length; i++)
    System.out.print(", " + a[i]);
  System.out.println(" }");
}

public static void load(double[] a, int start, int range)
{ int n=a.length;
  for (int i=0; i<n; i++)
    a[i] = random.nextInt(range) + start;
}
 public static void print(double[] a)
{ for (int i=0; i<a.length; i++)
    System.out.print(" " + (i>9?"":" ") + i);
  System.out.print("\n{ " + a[0]);
  for (int i=1; i<a.length; i++)
    System.out.print(", " + a[i]);
  System.out.println(" }");
}

public static void load(Object[] a, int start, int range)
{ int n=a.length;
  for (int i=0; i<n; i++)
  { int m = random.nextInt(range) + start;
    if (i%3==0) a[i] = new Integer(m);
    else if (i%3==1) a[i] = new Double(m);
    else a[i] = "(" + m + ")";
  }
}

public static void print(Object[] a)
{ for (int i=0; i<a.length; i++)
    System.out.print(" " + (i>9?"":" ") + i);
  System.out.print("\n{ " + a[0]);
  for (int i=1; i<a.length; i++)
    System.out.print(", " + a[i]);
```

```
    System.out.println(" }");
  }

  public static void print(String label, Object[] a)
  { System.out.print(label + " = ");
    for (int i=0; i<a.length; i++)
      System.out.print(" " + (i>9?"":" ") + i);
    System.out.print("\n{ " + a[0]);
    for (int i=1; i<a.length; i++)
      System.out.print(", " + a[i]);
    System.out.println(" }");
  }

  public static void swap(int[] a, int i, int j)
  { int temp = a[i];
    a[i] = a[j];
    a[j] = temp;
  }

  public static void swap(Object[] a, int i, int j)
  { Object temp = a[i];
    a[i] = a[j];
    a[j] = temp;
  }
  }
```

下面的测试类中仅有3个方法调用，其中两个load()方法用于变量及数组的初始化，在初始值的生成中利用了随机数产生器。而print()方法用于打印整型数组。load()方法将整数值限制在start到（start+range-1）之间，这有利于重复数值的产生。print()方法打印出一行数组下标值，对数组元素进行注解，有利于对索引元素的单个定位。

下面的测试驱动程序说明了该类是如何工作的。

```
Import schaums.dswj.Arrays;

public class Testing
   { private static final int SIZE = 16;
     private static final int START = 40;
     private static final int RANGE = 20;
     private static int[] a = new int[SIZE];

  public static void main(String[] args)
  { Arrays.load(a,START,RANGE);
    Arrays.print(a);
    Arrays.load(a,START,RANGE);
```

标准分享网 www.bzfxw.com 免费下载

```
    Arrays.print(a);
  }
}
```

该程序的输出为：

```
  0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
{ 49, 56, 46, 43, 40, 57, 47, 43, 43, 43, 46, 57, 47, 53, 44, 46}
  0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
{ 58, 55, 40, 45, 56, 46, 59, 59, 57, 45, 46, 42, 52, 47, 42, 54}
```

上述程序两次产生有 16 个元素的随机数数组，元素的随机数范围为 40 到 59，因而，存在重复的元数。例如，在第一次产生的数组中，a[2]=a[10]=46 就是如此。

2.4 顺序查找算法

顺序查找（Sequential Search）也称为线性查找（Linear Search）是最为简单的查找算法形式，当然，它的查找效率也是最低的。从开始元素起，它按顺序的方式依次检查每个元素，直到找到要查找的元素或到达数组末尾为止。

假如你在一辆行驶的列车上寻找某人时，你也许会使用顺序查找的方式来进行寻找，如图 2.1 所示。



图 2.1 顺序查找

算法 2.1 顺序查找

前提条件（输入）：

$s=\{ s_0,s_1,s_2,\ldots,s_{n-1} \}$ 是 n 个类型为 x 的有序值序列。

后置条件（输出）当某个 $s_i = x$ 时，返回索引值 i，否则，返回 -1.

1. 重复步骤 2～3 n 次，从 i 为 0 到 n-1。
2. (不变式) 子序列$\{s_0 \ldots s_{i-1}\}$中没有与 x 相等的元素。
3. 假如 $s_i = x$，则返回 i。
4. 返回 -1。

在算法 2.1（以及本书中的所有其他算法）中，我们使用前提和后置条件来说明算法的功能，并用循环不变式来证明算法的正确性。对每个算法均给出 Java 代码实例。

例 2.5 顺序查找

```
public static int sequentialSearch(int[] a, int x)
 { // 前提条件: returns i; if i >= 0, then a[i] == x;
  //                 otherwise i == -1;
  for (int i=0; i<a.length; i++)                          // 步骤 1
    // 不变式: if a[k]==x then i <= k < a.length;         // 步骤 2
```

```
    // 不变式: a[k] != x, for 0 <= k < i;                    // 步骤 2
    if (a[i]==x) return i;                                   // 步骤 3
  return -1;                                                 // 步骤 4
}
```

定理 2.1 顺序查找算法是正确的

证明：当n=0时，序列为空，所以循环体将不会被执行，只有步骤4将被执行，立即返回-1。此时它满足后置条件：x不等于数组中的任意一个元素，因为数组为空时它没有任何元素。

当n=1时，循环迭代一次，此时i=0，在此次循环中，$s_0 = x$ 或 $s_0 \neq x$。当 $s_0 = x$ 时，返回0，并满足后置条件。当 $s_0 \neq x$ 时，循环将被终止，步骤4将被执行，返回-1。此时它满足后置条件：序列中的那个单一元素不等于x。

假如n>1，则在循环中的第一次迭代中，i=0，子序列 $\{s_0..s_{i-1}\}$ 为空，所以第二步中的循环不变式自然为真。在第三步时，$s_0 = x$ 或 $s_0 \neq x$，当 $s_0 = x$ 时，返回0，并满足后置条件。当 $s_0 \neq x$ 时，循环将被继续进行。当存在第二次迭代时（也就是说 $s_0 \neq x$ 时），则i=1，由于子序列 $\{s_0..s_{i-1}\}$ 等于 $\{s_0\}$ 并且 $s_0 \neq x$，所以第二步中的不变式再次为真。

假设在循环迭代的第k次中，循环不变式为真，即子序列 $\{s_0..s_{k-1}\}$ 中没有一个元素等于x，则在该次迭代中，在第三步时，$s_i = x$ 或 $s_i \neq x$。当 $s_i = x$ 时，返回k，并满足后置条件。当 $s_i \neq x$ 时，循环将被继续进行。

由数学归纳原理（见附录A.4）可知，在每一循环迭代之后，算法要么在后置条件为真的情况下终止，要么用于下次迭代的循环不变式为真。因此，假如在任意步中没有终止，则在最后迭代之后（当i=n-1），对于i=n次迭代时，用于该次迭代的循环不变式为真，也就是说，子序列 $\{s_0..s_{n-1}\}$ 中没有任意元素等于x。此时，将返回-1，并满足后置条件。

下面的定理中使用了O（）记号，见附录A.3中的复习部分。

定理 2.2 顺序查找算法的运行时间是O（n）

证明：如果x位于序列中，假定在i处，$x = s_i$，且 $i < n$，则循环将迭代i次。在这种情况下，运算时间正比于i，即为O（n）（因为i<n）。

假如x不在序列中，循环将迭代n次，使得运行时间正比于n，也就是O（n）。

例 2.6 测试顺序查找算法

```
import schaums.dswj.Arrays;

public class Testing
 { private static final int SIZE = 15;
   private static final int START = 40;
   private static final int RANGE = 20;
   private static int[] a = new int[SIZE];
```

标准分享网 www.bzfxw.com 免费下载

```
    public static void main(String[] args)
    { Arrays.load(a,START,RANGE);
      Arrays.print(a);
      test( );
      test( );
      test( );
      test( );
    }

    public static void test( )
    { int x = Arrays.load(START,RANGE);
      System.out.print("Searching for x - " + x + ":\t");
      int i = Arrays.sequentialSearch(a,x);
      if (i >= 0) System.out.println("a[" + i + "] = " + a[i]);
      else System.out.println("i = " + i + " ==> x not found");
    }
}
```

该程序的输出结果为：

```
  0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15
{ 49, 44, 49, 41, 50, 59, 51, 48, 48, 41, 50, 47, 41, 58, 46, 48}
Searching for x = 51:  a[ 6 ] = 51
Searching for x = 47:  a[ 11 ] = 47
Searching for x = 56:  i = -1==> x not found
Searching for x = 50:  a[ 4 ] = 50
```

本程序测试顺序查找算法，在由 load()产生的数组上进行 4 次查找。第一次查找 x = 51，并且在 a[6]处找到该元素。第二次查找 x = 47，并在 a[11]处找到该元素。第三次查找 x = 56，但失败，返回-1。第四次查找 x = 50，并在 a[4]处找到该元素。注意该值也存在于 a[10]处。顺序查找算法一旦找到目标，就放弃对数组中其余部分的查找。

为了访问的方便，每个数组查找算法均加入到 achaums.dswj.Arrays 类中。例如，对顺序查找算法的测试，我们会将该类引入，然后调用 Arrays.sequentialSearch()即可。

2.5 折半查找算法

折半查找（Binary Search）是查找有序序列的标准过程，比顺序查找有效得多，但它需要元素是有序的。它把查找序列平均分成两半，如划分点元素不是我们所要查找的元素，则在有可能包含查找元素的子序列中查找，重复上述查找过程，直到找到或找遍为止，参见图 2.2。

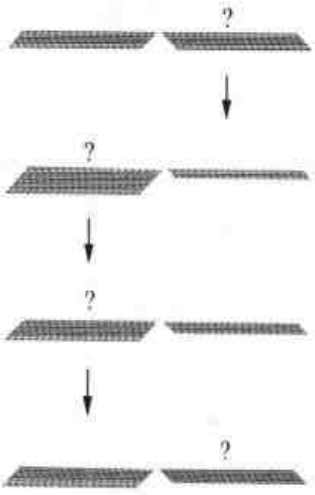


图 2.2 折半查找

当你查字典时，或许你使用的就是折半查找。

算法 2.2 折半查找

前提条件（输入）：s={ $s_0,s_1,s_2,\ldots,s_{n-1}$ } 是 n 个类型为 x 的有序值序列。

后置条件（输出）当某个 $s_i = x$ 时，返回索引值 i，否则，返回 -1。

1. 令 ss 为序列 s 的子序列，并置初值为 s。
2. 当子序列 ss 为空时，返回-1。
3. (不变式) 当原序列 s 中包含 x 时，x 必然在子序列 ss 中。
4. 令 s_i 为子序列 ss 的中间元素。
5. 若 $s_i = x_i$ 则返回索引 i 。
6. 假如 $s_i < x$，重复步骤 2～7，在子序列{$s_{i+1}\ldots s_{n-1}$}中继续查找与 x 相等的元素。
7. 重复步骤 2～7，在子序列{$s_0\ldots s_{i-1}$}中查找与 x 相等的元素。

注意，在算法 2.2 的前提条件中，序列必需是事先排序的。

在 java.util.Arrays 中实现了折半查找算法（见定理 2.3），这里给出它的一个注释版本。

例 2.7 折半查找算法

```
public static int binarySearch(int[] a, int x)
 { // 前置条件:  a[0] <= a[1] <= ... <= a[a.length-1];
   // 后置条件:  returns i; if i >= 0, then a[i] == x;
  //              otherwise i == -1;
  int lo=0, hi=a.length-1;
   while (lo <= hi)                                  // 步骤 1
  { // 不变式: if a[j]== x then lo <= j <= hi;      // 步骤 3
    int i = (hi + lo)/2;                             // 步骤 4
    if (a[i] == x) return i;                         // 步骤 5
    else if (a[i] < x) lo = i+1;                     // 步骤 6
    else hi = i-1;                                   // 步骤 7
   }
  return -1;                                         // 步骤 2
}
```

定理 2.3 折半查找算法是正确的

证明：循环迭代开始时，子序列为输入原序列，循环不变式为真。在随后的各循环迭代步中，下一迭代步中的子序列是当前子序列中去掉不包含元素 x 的那半段后所剩余的部分，因此，如果 x 在原序列中，则 x 必在下一迭代步中将要查找的子序列中，因此，在每一循环步中，不变式总为真。

在每次迭代中，当 $s_i = x$ 时，将返回索引值 i。否则，子序列长度将减少百分之五十多。因原序列为有限个元素，循环必在有限步内终止，因此，当算法终止于循环中部（步骤 5）

标准分享网 www.bzfxw.com 免费下载

时，将返回索引值 i。或当终止于循环末尾（步骤 6 或 7）时，将返回-1。从循环中部返回索引值 i 时，表明 $s_i = x$，否则，循环终止于 hi<lo，也即当前子序列为空序列。在这种情况下，从循环不变式可知，s_i 不在原序列中。

定理 2.4 折半查找算法的运行时间为 O(lg n)

证明：从定理 2.3 的证明过程可知，循环迭代次数最多为 n 能连续被 2 除（取整）的次数加 1，也即 n 以 2 为底的对数取整加 1，因此，算法的时间复杂性为 O(lg n)。

例 2.8 测试折半查找算法

```
import schaums.dswj.Arrays;

public class Ex0208
{ private static final int SIZE = 16;
  private static final int START = 40;
  private static final int RANGE = 20;
  private static int[] a = new int[SIZE];

  public static void main(String[] args)
  { Arrays.load(a,START,RANGE);
    Arrays.print(a);
    test( );
    java.util.Arrays.sort(a);
    Arrays.print(a);
    test( );
    test( );
    test( );
  }

  public static void test( )
  { int x = Arrays.load(START,RANGE);
    System.out.print("Searching for x = " + x + ":\t");
    int i = Arrays.binarySearch(a,x);
    if (i >= 0) System.out.println("a[" + i + "] = " + a[i]);
    else System.out.println("i = " + i + " --> x not found");
  }
}
```

该程序的输出为：

```
  0   1   2   3   4   5   6   7   8   9   10   11  12  13  14  15
{ 48, 57, 46, 51, 55, 51, 55, 45, 47, 52, 57, 47, 50, 42, 59, 45}
Searching for x = 50:  i = -1 ==> x not found
```

```
  0   1   2   3   4   5   6   7   8   9   10   11   12   13   14   15
{ 42, 45, 45, 46, 47, 47, 48, 50, 51, 51, 52, 55, 55, 57, 57, 59}
Searching for x = 50:  a[ 7 ] = 50
Searching for x = 47:  a[ 5 ] = 47
Searching for x = 58:  i = -1 ==> x not found
```

本测试程序中，test()被调用 4 次，从程序中可以看出，test()调用折半查找算法在由 load() 产生的数组中查找 x，第一次查找 x=50，尽管数组中 a[7]= 50，但由于数组没有事先排序，因此折半查找失败，没有找到。在第二次调用 test()之前，使用 java.util.Arrays.sort(a)对数组 a 进行排序，这样，在 a[7]找到 x=50。第三次查找时，x=57，在 a[5]处找到。请注意的是，此处并不是 x=57 的第一次出现，因为它还出现在 a[4]处。由于折半查找算法不是顺序的，当某值不只一次出现时，算法返回的索引值是不易预见的（见复习题 2.9）。

第四次调用 test()查找 x=58，没有找到，返回值为-1。

2.6 Vector 类

从数学上讲，向量（Vector）就是数字的一个有穷序列，从几何上看，我们可以把二维向量（x , y）看成平面中的一个点，而三维向量（x , y , z）表示三维空间中的一个点。在线性代数中，经常可见到 n 个元素的向量(x_1 , x_2 , ... x_n)，并使用下标来标识某个元素。

在 Java 中，除以下两点外，向量与数组完全相同。

- 一个向量是类 java.util.Vector 的实例。
- 一个向量的长度可以改变。

在 Java1.2 中，Vector 类几乎是重写的，它定义在 java.util 包中，类定义为：

```
public class Vector extends AbstractList implements List
{
  public boolean     add( Object Object)
  public void        add( int index, Object object)
  public boolean     addAll( Collection collection)
  public boolean     addAll( int index, Collection collection )
  public void        addElement(Object object)
  public void        clear( )
  public Object      clone( )
  public boolean     contains( Object object)
  public boolean     containsAll(Collection collection)
  public void        copyInto(Object[ ] objects)
  public Object      elementAt(int index)
  public boolean     equals(Object object)
  public Object      firstElement( )
  public Object      get(int index)
```

标准分享网 www.bzfxw.com 免费下载

```
public int          hashCode( )
public int          indexOf(Object object)
public int          indexOf(Object object, int index)
public void         insertElementAt(Object object , int index)
public boolean      isEmpty( )
public Object       lastElement( )
public int          lastIndexOf(Object object)
public int          lastIndexOf(Object object, int index)
public Object       remove(int index )
public boolean      remove(Object object)
public boolean      removeAll(Collection collection)
public void         removeAllElements( )
public boolean      removeElement(Object object)
public void         removeElementAt(int index)
public boolean      retainAll(Collection collection)
public Object       set(int index, Object object)
public void         setElementAt(Object object, int index)
public int          size( )
public List         subList(int start, int stop)
public Object[ ]    toArray( )
public Object[ ]    toArray(Object[ ] objects)
public String       toString( )
public              Vector( )
public              Vector(Collection collection)
}
```

其中，add 方法用于将新元素插入到向量之中，add(object) 和 addElement(object)方法具有相同的作用，区别在于 add(object) 有返回值，操作成功时，返回 true，否则返回 false。当没有指明要插入的新元素的位置时，缺省插入到向量的末端。

方法 clear()使得向量为空，从而使向量的尺寸为 0。注意一个空向量与一个空引用是不一样的，一个空向量仍是一个向量，向量对象仍然存在。

方法 clone()产生该向量的一个完整的复制品，因而向量所包含的对象也得到克隆。

方法 contains 用于确定某个对象是否是该向量的成员。

方法 copyInto 将向量中的元素拷贝到一个指定的数组。

方法 elementAt()和 get()与数组中的下标算子类似，返回给定索引位置的向量元素。

方法 equals()进行向量对象的相等性测试，当两对象均为向量对象，且两向量中的对应元素均相等时，将返回真值。

方法 firstElement()和 lastElement()分别返回向量中的第一个和最后一个元素。

方法 hashCode()为该向量对象返回一个标识数。

方法 indexof 使用顺序查找算法在向量中查找某个给定的对象，当在向量中找到该对象时，就返回该对象在向量中的位置索引，否则返回 -1。当该方法带两个参数时，就从指定

的索引位置往后查找。

方法 InsertElementAt()在指定位置将对象插入到向量之中，并使得前面的元素往前挪动。

方法 lastIndexOf 方法与 indexof 方法的不同之处在于该方法中的顺序查找算法的查找方向正好相反。

方法 remove 从向量中去掉一到多个元素，并使右边的剩余元素往左挪动，该方法的效果与 insertElementAt()方法正好相反。应当注意，用于插入元素的方法只有一个，但有 6 个 remove 方法，方法 remove(index)和 removeElementAt(index)除在返回值方面不同外，其操作效果是一样的，前者在操作成功时返回去掉的对象（否则为 null），而后者在操作成功时返回真值，没有成功时为假。

方法 set(index, object)和 setElementAt(object, index)均用给定的对象去替代 index 所标识的对象，这两个方法除了前者返回被替代的对象（操作不成功时返回 null）外，具有相同的操作效果。

方法 subList()返回一个 list 对象，该对象中的元素为向量中从 start 到 stop – 1 的 stop – start 个元素。

方法 toArray()返回一个 Object[] 数组，该数组中所包含的元素与向量中的一致。不带参数的 toArray()方法产生一个数组，并返回该数组，它的尺寸与向量相同。而带 Object 参数的 toArray()方法在数组尺寸小于向量尺寸时与不带参数的方法是一样，但当数组尺寸大于向量时，将向量中的元素拷入到数组中的前部，并使数组中的其余元素为空。

方法 toString()返回一个字符串，用于表示向量的内容。

例 2.9 测试 java.util.Vector 类

```
import java.util.*;

public class Ex0209
{ private static Vector v = new Vector( );
  private static Vector w = new Vector( );
  public static void main(String[] args)
  { String[] cities = { "Austin", "Boston", "Fresno", "Toledo" };
    v.addAll(Arrays.asList(cities));
    System.out.println("v = " + v);
    v.add("Tucson");
    System.out.println("v = " + v);
    w = (Vector)v.clone( );
    System.out.println("w = " + w);
    System.out.println("w.equals(v) = " + w.equals(v));
    v.set(3,"Ottowa");
    System.out.println("v = " + v);
    System.out.println("w = " + w);
```

标准分享网 www.bzfxw.com 免费下载

```
        System.out.println("w.equals(v) - " + w.equals(v));
        v.insertElementAt("London",3);
        System.out.println("v = " + v);
        System.out.println("w = " + w);
        System.out.println("w.equals(v) = " + w.equals(v));
        w.removeElementAt(1);
        w.removeElementAt(3);
        w.remove("Fresno");
        System.out.println("w = " + w);
        v.addAll(5,w);
        System.out.println("v = " + v);
        System.out.println("v.indexOf(\"Austin\") = "
                           + v.indexOf("Austin"));
        System.out.println("v.indexOf(\"Austin\",2) = "
                           + v.indexOf("Austin",2));
        System.out.println("v.indexOf(\"Dublin\") = "
                           + v.indexOf("Dublin"));
    }
}
```

该程序的输出为：

```
v = [Austin, Boston, Fresno, Toledo]
v = [Austin, Boston, Fresno, Toledo, Tucson]
w = [Austin, Boston, Fresno, Toledo, Tucson]
w.equals(v) = true
v = [Austin, Boston, Fresno, Ottowa, Tucson]
w = [Austin, Boston, Fresno, Toledo, Tucson]
w.equals(v) = false
v = [Austin, Boston, Fresno, London, Ottowa, Tucson]
w = [Austin, Boston, Fresno, Toledo, Tucson]
w.equals(v) = false
w = [Austin, Toledo]
v = [Austin, Boston, Fresno, London, Ottowa, Austin, Toledo, Tucson]
v.indexOf("Austin") = 0
v.indexOf("Austin", 3 ) = 5
v.indexOf("Dublin") = -1
```

数组 cities 用于生成向量 v，通过方法 Arrays.asList()调用使得该数组成为 List 对象，并调用带 Collection 参数（List 是 Collection 的子类）的 Vector 构造器产生向量 v。System.out.println()方法通过调用向量的 toString()方法来向量内容进行输出。

通过调用 v.add("Tucson")将该字符串添加到向量末尾，从而使向量尺寸加 1。而方法 v.clone()返回向量 v 的克隆，但作为一个 Object 类型，因此，我们在用类型转换

符（Vector）使之转换为Vector类型，并将其赋值给引用w。

方法equals()表明v和w是相等的，而调用v.set(3, "Ottowa")将改变引用v所标识的对象，而不会改变w所标识的对象，从而表明这两个向量确实是独立的而且相等。

从w中去掉3个元素后，调用v.addAll(5, w)将w中从第五个元素开始的所有元素的拷贝插入到v中，最后，indexOf()方法用于查找v中的元素。

复 习 题

2.1 数组中的基元与成分的不同是什么？

2.2 说Java不允许多维数组时指的是什么？

2.3 异常ArrayIndexOutOfBoundsException是什么？它在使Java区别于其他语言如C和C++方面有什么作用？

2.4 数组索引的类型可以是哪些？

2.5 以下定义中有什么错误？

```
Arrays arrays= new Arrays( );
```

2.6 打印一个对象数组的简单方法是什么？

2.7 如果说折半查找远快于顺序查找，为什么后者还会被使用呢？

2.8 在数组中，当某元素重复出现时，使用顺序查找会发生什么情况？

2.9 在数组中，当某元素重复出现时，使用折半查找会发生什么情况？

2.10 对一个向量对象调用clear()方法与将该对象引用设置为null有什么不同？

习 题

2.1 运行一个测试程序，看Arrays.fill()方法是如何处理一个数组对象的？

2.2 运行一个测试程序，看Arrays.equals()方法是如何处理一个数组对象的？

2.3 运行一个测试程序，看Arrays.equals()方法是如何处理一个以数组为成分的数组对象的？

2.4 当顺序查找对10 000个元素的数组所需时间为50毫秒时，在同样的机器上对20 000个元素的数组进行查找时，你预计所需时间为多长？

2.5 当折半查找对1 000个元素的数组所需时间为5毫秒时，在同样的机器上对1 000 000个元素的数组进行查找时，你预计所需时间为多长？

2.6 编写并运行一个调用例2.7中折半查找的一个驱动程序。

2.7 运行一个调用例2.7中折半查找的一个测试驱动程序，假定数组包含10 000个元素，并计算循环迭代步。

标准分享网 www.bzfxw.com 免费下载

2.8 编写并测试以下方法:

```
private static boolean isSorted( int [ ] a)
// 当且仅当 a[0] <= a[1] <= … <=a[a.length -1 ]时返回真
```

2.9 编写并测试以下方法:

```
private static int minimum ( int [ ] a)
// 返回数组 a[ ]中的最小元素
```

2.10 编写并测试以下方法:

```
private static int double mean( double [ ] a)
// 返回数组 a[ ]中的所有元素的平均值
```

2.11 编写并测试以下方法:

```
private static int[ ] withoutDuplicates( int [ ] a)
// 返回一个数组，该数组的元素与 a[ ]中的相同，但没有重复元素
```

2.12 编写并测试以下方法:

```
private static Object[ ] withoutDuplicates( Object [ ] a)
// 返回一个数组，该数组中的成份与数组 a[ ]中的相同，但没有重复
```

2.13 编写并测试以下方法:

```
private static void reverse( int [ ] a)
//将数组 a[ ]中的元素前后倒置，即最前的放到最后，第二个放到倒数第二个，等等
```

2.14 编写并测试以下方法:

```
private static Object[ ] concatenate( Object [ ] a, Object [ ] b)
// 返回一个数组，该数组中的成份的前部分与数组 a[ ]中的相同，后面的部分与数组 b[ ]中的
// 相同
```

2.15 编写并测试以下方法:

```
private static void shuffle( Object [ ] a)
// 对数组中的成份进行随机性序列改变
```

2.16 编写并测试以下方法:

```
private static int[ ] tally( String string)
// 返回一个有 26 个整数值的数组 a，该数组记录了字符串 string 中各字母（不区分大小写）的
// 出现频率
```

2.17 编写并测试以下方法:

```
private static double innerProduct(double[ ] x, double [ ] y)
// 将两数组 x 和 y 看成向量，该方法返回这两个向量的内积，即两向量各对应分量的乘积之和
```

2.18 编写并测试以下方法:

```
private static double outerProduct( double[ ] x, double [ ] y)
//将两数组x和y看成向量,该方法返回这两个向量的外积,在结果数组中,p[i][j] = a[i]*y[j]
```

2.19 编写并测试以下方法：

```
private static double[ ] [ ] product( double [ ] [ ] a, double [ ] [ ] b)
//将两个数组看成矩阵，该方法返回两矩阵的乘积， p[i][j] = Sum( a[i][k]* b[k][j] :k)
```

2.20 编写并测试以下方法：

```
private static void transpose(double[ ] [ ]  a)
// 将数组看成矩阵，该方法对矩阵进行转置：a[ i ][ j ] <-- a[j][i]
```

2.21 编写并测试以下方法：

```
private static int[ ] [ ]  pascal( int size )
// 返回给定大小的帕斯卡三角形
```

2.22 Eratosthenes 筛是一个布尔元素的数组，其中第 i 个元素为真当且仅当 i 是一个素数。使用以下算法计算并等于大小为 1 000 的 Eratosthenes 筛。

算法 2.3 Eratosthenes 筛

（前置条件：p 是一个 n 位数组。）

（后置条件：当且仅当 i 是一个素数时 p[i]为真。）

1. 将 $p[0]$和 $p[1]$初始化为假，并使其他 $p[i]$为真。
2. 对从 3 到 n 中的每个 i（步长为 2），重复步骤 3。
3. 如果存在一个小于等于 i 的平方根的素数可以整除 i，则将 $p[i]$置为假。

2.23 使用 java.util.Vector 来重新求解习题 2.22。

2.24 使用 java.util.BitSet 来重新求解习题 2.22。

2.25 用以下方法定义并测试 Primes 类：

```
private Primes( )
public static void setLast( int last )     // sets last
public static void setLast( )              // sets last = 1
public static void sizeSize( int size )    // sets size of bitsize
public static void sizeSize( )             //sets bitset size = 1000
public static boolean isPrime( int n )     // true if n is prime
public static int nextPrime( )             // next prime after last
public static void printPrime( )           // prints sieve
```

使用问题 2.24 中采用 BitSet 的 Eratosthenes 筛实现方法，并应用以下定义：

```
public class Primes
{ private static final int SIZE = 1000;
  private static int size = SIZE;
  private static BitSet sieve = new BitSet( size );
```

标准分享网 www.bzfxw.com 免费下载

```
private static int last = 1;
```

包括以下静态测试化代码，它用于实现 Eratosthenes 筛。

```
Static
{ for ( int i=2; i<SIZE; i++)
   sieve.set(i );
  for( int n=2; 2*n<SIZE; n++)
   if (sieve.get(n ))
     for ( int m=n; m*n <SIZE; m++)
       sieve.clear( m*n );
}
```

2.26 将下面的方法加入到 Primes 类中，并对它进行测试。

```
public static String factor( int n)
// 前置条件：n > 1
// 返回 n 的素数因子；
// 例如，factor( 4840 )返回"2*2*2*5*11*11"
```

2.27 哥德巴赫(Christian Goldbach)（1690～1764）在 1742 年提出任何大于 2 的偶数均为两个素数之和的猜想。编写一个程序对小于 100 的所有偶数进行测试，来检验哥德巴赫猜想。利用习题 2.25 中的 Primes 类，输出中的前 10 行应当如下所示：

```
4 = 2+2
6 = 3+3
8 = 3+5
10 = 3+7 = 5+5
12 = 5+7
14 = 3+11 = 7+7
16 = 3+13 = 5+11
18 = 5+13 = 7+11
20 = 3+17 = 7+13
22 = 3+19 = 5+17 = 11+11
```

2.28 皮埃里·费马(Pierre de Fermat)（1601～1665）曾猜想对某个整数 p 而言，存在形为 $n=2^{2^p}+1$ 的素数，这些数称为费马素数，费马素数有无穷多个。例如，5 是一个费马素数，因为它是一个素数，并且具有 2^{2^p} 的形式。编写一个程序用于找出 int 类型范围中的所有费马素数。使用 2.25 中定义的 Primes 类以及 Math.pow 方法，输出中的前 5 行应如下所示：

```
2^2^0 + 1 =3
2^2^1 + 1 =5
2^2^2 + 1 =17
2^2^3 + 1 =257
2^2^4 + 1 =65 537
```

2.29 查里斯·巴比奇(Charles Babbage)（1792～1871）在1823年获得历史上首次政府同意的研究计划，他说服英国政府提供1000英镑用于建立它的差分机器，在获准的建议中，巴比奇把 $x^2 + x + 41$ 作为计算机的制表函数。该函数对数学家来讲具有特别的意义，因为它将产生一个非同寻常的素数。具有形为 $n = x^2 + x + 41$ 的素数（对于某个数x）称为巴比奇素数。编写一个程序，找出所有小于10 000的巴比奇素数。使用2.25中的定义的Primes类，输出中的前5行应如下所示：

```
0          41 is prime
1          43 is prime
2          47 is prime
3          53 is prime
4          61 is prime
```

2.30 两个紧靠在一起的奇数如果均为素数，则称它们为孪生素数。孪生素数猜想是：存在无穷多对孪生素数。编写一个程序，找出所有小于1 000的孪生素数。使用2.25中的定义的Primes类，输出中的前5行应如下所示：

```
3          5
5          7
11         13
17         19
29         31
```

2.31 测试如下猜想：一对紧靠在一起的平方数间至少存在一个素数（平方数为1，4，9，16，25，…）。使用2.25中的定义的Primes类，输出中的前5行应如下所示：

```
1<2<4
4<5<9
9<11<16
16<17<25
25<29<36
```

2.32 米尼密特·弗里埃马林·梅森尼（Minimite friar Marin Mersenne）(1588～1648)在1644年从事形为 $n = 2^p - 1$ 的数的研究（p为素数），他相信这些数中的大多数也是素数，现在称这样的数为梅森尼素数。编写一个程序，找出所有 $p < 30$ 的梅森尼素数。使用2.25中的定义的Primes类，输出中的前5行应如下所示：

```
2          2^2 - 1 = 3  is prime
2          2^3 - 1 = 7  is prime
5          2^5 - 1 = 31  is prime
```

标准分享网 www.bzfxw.com 免费下载

```
7        2^7 - 1 = 127  is prime
11       2^11 - 1 = 2047  is not  prime
```

2.33 一个数的数位在进行前后颠倒时不变时称为回文数，例如，3456543 就是一个回文数。编写一个程序，对前 10 000 个素数进行检查，将是回文数的素数进行输出。使用 2.25 中的定义的 Primes 类。

复习题答案

2.1 数组的成份可以是任何类型：基本类型、引用或数组。一个数组元素是本身不是数组类型的成份。因此，对 a [] [] 而言，数组 a [] 的成份为行数组，而 a [] []的元素为 double 变量。

2.2 多维数组是一个有 2 个或 2 个以上下标值的数组。一个 Java 数组是只有一个下标变量的数组。然而，变量所标识的是一个本身又是数组的成份（有一个下标），因此，原数组似乎有多于一个的下标。

2.3 ArrayIndexOutOfBoundsException 对象是一个异常，该异常在用小于 0，大于或等于数组长度的下标值对数组进行访问时发生。这给程序员对运行时的错误控制提供了某种手段，在如 C++的程序设计语言中，这样的错误往往导致程序崩溃。

2.4 数组的下标类型可以是 byte、char、short 或 int。

2.5 Array 类不能被实例化，这是由于它的构造函数是私有的（private）。

2.6 打印对象数组的最简单的方式是将它作为参数传递给 Arrays.toList()方法，它将对象数组转换为一个 List 对象，该 List 对象可以用 System.out.println() 方法进行直接打印输出。

2.7 折半查找将不能工作，除非首先对序列进行排序。

2.8 如果顺序查找算法用于查找数组中某个不止出现一次的元素，它将返回最靠近数组开始位置的元素的下标。

2.9 如果折半查找算法用于查找数组中某个不止出现一次的元素，它可能返回这些重复元素中的任意一个的下标，它取决于离子区间中心点的远近。例如，如果折半查找算法用于查找一个有 10 000 个元素的数组，查找一个位于 0 到 99 间的重复值，查找算法将在第 7 次迭代中返回下标索引值 77。

2.10 方法调用 v.clear()将使向量 v 成为一个空向量，空向量仍是一个非空（non-null ）对象，但赋值语句 v = null 将使得对象引用为空，并使得原有对象消灭。在前一种情况下，表达式 v.size() 的值为 0，而在后一种情况下，表达式 v.size()将导致异常的产生。

习 题 解 答

2.1 测试用于对象数组的java.util.Arrays.fill()方法。

```
import java.util.Arrays;
public class Pr0201
{ public static void main(String[] args)
  { Object[] a = new Object[4];
    Double x = new Double(Math.PI);
    Arrays.fill(a,x);
    for (int i=0; i<a.length; i++)
      System.out.println("a[" + i + "] = " + a[i]);
    Arrays.fill(a,"Cherry Pie!");
    for (int i=0; i<a.length; i++)
      System.out.println("a[" + i + "] = " + a[i]);
  }
}
```

2.2 测试用于对象数组的java.util.Arrays.equals()方法。

```
import java.util.Arrays;

public class Pr0202
{ public static void main(String[] args)
  { Double x = new Double(Math.PI);
    Object[] a = new Object[4];
    Arrays.fill(a,x);
    Object[] b = new Object[4];
    Arrays.fill(b,x);
    System.out.println("b.equals(a) = " + b.equals(a));
    System.out.println("Arrays.equals(a,b) = "
                       + Arrays.equals(a,b));
    Arrays.fill(a,"Cherry Pie!");
    Arrays.fill(b,"Cherry Pie!");
    System.out.println("b.equals(a) = " + b.equals(a));
    System.out.println("Arrays.equals(a,b) = "
                       + Arrays.equals(a,b));
  }
}
```

2.3 测试用于对象数组的java.util.Arrays.equals()方法。

```
import java.util.Arrays;
import java.util.Vector;
public class Pr0203
```

标准分享网 www.bzfxw.com 免费下载

```
{ public static void main(String[] args)
  { double[] x = { Math.E, Math.PI };
    String[] s = { "North", "East", "South", "West" };
    Vector[] y = new Vector[0];
    Object[] a = { x, s, null, y };
    Object[] b = { x, s, null, y };
    System.out.println("b.equals(a) = " + b.equals(a));
    System.out.println("Arrays.equals(a,b) = " + Arrays.equals(a,b));
  }
}
```

2.4 顺序查找算法的运行时间是线性时间，也即它的运行时间与数组中元素的个数成正比。因而一个元素个数为两倍的数组所需的执行时间也为两倍，所以时间为20ms。

2.5 折半查找算法的运行时间为对数时间，因而问题规模的平方仅使它所需的时间加倍。因此，一个有$1\,000^2$个元素的数组所需的处理时间将加倍，即为10ms。

2.6 插补算法的运行时间为超指数时间，即使问题规模成为原来的平方，对执行时间也没有太大的影响，因此，一个具有1 000 000个元素的数组所需的处理时间大概也为2ms。

2.7 测试顺序查找算法。

```
public class Pr0207
{ private static final int N=16;
  private static int[] a = new int[N];
  private static final int RANGE=2*N;
  private static final int START=10;
  private static Random random = new Random( );
  public static void main(String[] args)
  { load(a);
    print(a);
    int x = random.nextInt(RANGE) + START;
    System.out.println("x = " + x);
    int i = sequentialSearch(a,x);
    System.out.println("search(a,x) = " + i);
    if (i >= 0) System.out.println("a[" + i + "] = " + a[i]);
  }
  private static void load(int[] a)
  { for (int i=0; i<a.length; i++)
      a[i] = random.nextInt(RANGE) + START;
  }
  private static void print(int[] a)
  { for (int i=0; i<a.length; i++)
      System.out.print(" " - (i>9?"":" ") + i);
    System.out.print("\n[ " + a[0]);
    for (int i=1; i<a.length; i++)
```

```
      System.out.print(", " + a[i]);
    System.out.println(" }");
  }
  public static int sequentialSearch(int[] a, int x)
  { // Sequential Search:
    for (int i=0; i<a.length; i++)
      if (a[i]==x) return i;
    return -1;
  }
}
```

2.8 测试 isSorted(int[])方法。

```
public class Pr0208
{ private static final int SIZE = 16;
  private static int[] a = new int[SIZE];
  public static void main(String[] args)
  { schaums.dswj.Arrays.load(a,40,20);
    schaums.dswj.Arrays.print(a);
    System.out.println("isSorted(a) = " + isSorted(a));
    java.util.Arrays.sort(a);
    schaums.dswj.Arrays.print(a);
    System.out.println("isSorted(a) = " + isSorted(a));
  }
  private static boolean isSorted(int[] a)
  { if (a.length<2) return true;
    for (int i=1; i<a.length; i++)
      if (a[i]<a[i-1]) return false;
    return true;
  }
}
```

2.9 测试 minimum(int [])方法。

```
import schaums.dswj.Arrays;
public class Pr0209
{ private static final int SIZE = 8;
  private static int[] a = new int[SIZE];
  public static void main(String[] args)
  { schaums.dswj.Arrays.load(a,20,80);
    schaums.dswj.Arrays.print(a);
    System.out.println("minimum(a) = " + minimum(a));
  }
  private static int minimum(int[] a)
  { int min = a[0];
    for (int i=1; i<a.length; i++)
```

标准分享网 www.bzfxw.com 免费下载

```
    if (a[i]<min) min = a[i];
   return min;
  }
}
```

2.10 测试 mean(double[])方法。

```
import schaums.dswj.Arrays;
public class Pr0210
{ private static final int SIZE = 4;
  private static double[] a = new double[SIZE];
  public static void main(String[] args)
  { schaums.dswj.Arrays.load(a,20,10);
    schaums.dswj.Arrays.print(a);
    System.out.println("mean(a) = " + mean(a));
  }
  private static double mean(double[] a)
  { double sum=0.0;
    for (int i=0; i<a.length; i++)
      sum += a[i];
    return sum/a.length;
  }
}
```

2.11 测试 withoutDuplicates(int [])方法。

```
import schaums.dswj.Arrays;
public class Pr0211
{ private static final int SIZE = 16;
  private static int[] a = new int[SIZE];
  public static void main(String[] args)
  { schaums.dswj.Arrays.load(a,10,8);
    schaums.dswj.Arrays.print(a);
    int[] b = withoutDuplicates(a);
    schaums.dswj.Arrays.print(b);
  }
  private static int[] withoutDuplicates(int[] a)
  { if (a.length<2) return a;  // there are no duplicates
    int x = a[0];  // use this value as a dummy marker
    for (int i=1; i<a.length-1; i++)
      for (int j=i+1; j<a.length; j++)
        if (a[j]==a[i]) a[j] = x;
    int count=0;  // count the duplicates
    for (int i=1; i<a.length; i++)
      if (a[i]==x) ++count;
    if (count==0) return a;  // there are no duplicates
```

```
    int[] b = new int[a.length-count];
    b[0] = x;
    int shift=0;
    for (int i=1; i<a.length; i++)
      if (shift>0 && a[i] != x) b[i-shift] = a[i];
      else if (a[i]==x) ++shift;
      else b[i] = a[i];
    return b;
  }
}
```

2.12 测试 withoutDuplicates(Object [])方法。

```
import schaums.dswj.Arrays;
public class Pr0212
{ private static final int SIZE = 15;
  private static Object[] a = new Object[SIZE];

  public static void main(String[] args)
  { schaums.dswj.Arrays.load(a,10,8);
    schaums.dswj.Arrays.print(a);
    Object[] b = withoutDuplicates(a);
    schaums.dswj.Arrays.print(b);
  }

  private static Object[] withoutDuplicates(Object[] a)
  { if (a.length<2) return a;  // there are no duplicates
    Object x = a[0];  // use this object as a dummy marker
    for (int i=1; i<a.length-1; i++)
      for (int j=i+1; j<a.length; j++)
        if (a[j].equals(a[i]))
        a[j] = x;
    int count=0;  // count the duplicates
    for (int i=1; i<a.length; i++)
      if (a[i].equals(x)) ++count;
    if (count==0) return a;  // there are no duplicates
    Object[] b = new Object[a.length-count];
    b[0] = x;
    int shift=0;
    for (int i=1; i<a.length; i++)
      if (shift>0 && !a[i].equals(x)) b[i-shift] = a[i];
      else if (a[i].equals(x)) ++shift;
      else b[i] = a[i];
    return b;
  }
```

标准分享网 www.bzfxw.com 免费下载

```
}
```

2.13 测试 reverse(int[])方法。

```
public class Pr0213
{ private static final int SIZE = 16;
  private static int[] a = new int[SIZE];
  public static void main(String[] args)
  { schaums.dswj.Arrays.load(a,60,40);
    schaums.dswj.Arrays.print(a);
    reverse(a);
    schaums.dswj.Arrays.print(a);
  }

  private static void reverse(int[] a)
  { if (a.length<2) return;
    for (int i=0; i<a.length/2; i++)
      schaums.dswj.Arrays.swap(a,i,a.length-1-i);
  }
}
```

2.14 测试 concatenate(Object[], Object[])方法。

```
public class Pr0214
{ private static final int SIZE = 8;
  private static Object[] a = new Object[SIZE];
  private static Object[] b = new Object[SIZE];
  public static void main(String[] args)
  { schaums.dswj.Arrays.load(a,10,40);
    schaums.dswj.Arrays.load(b,60,40);
    schaums.dswj.Arrays.print("a",a);
    schaums.dswj.Arrays.print("b",b);
    Object[] c = concatenate(a,b);
    schaums.dswj.Arrays.print("a",a);
    schaums.dswj.Arrays.print("b",b);
    schaums.dswj.Arrays.print("c",c);
  }

  private static Object[] concatenate(Object[] a, Object[] b)
  { Object[] c = new Object[a.length+b.length];
    for (int i=0; i<a.length; i++)
      c[i] = a[i];
    for (int i=0; i<b.length; i++)
      c[i+a.length] = b[i];
    return c;
  }
```

```
}
```

2.15 测试 shuffle(Object[])方法。

```
import java.util.Random;
public class Pr0215
{ private static final int SIZE = 16;
  private static Object[] a = new Object[SIZE];
  public static void main(String[] args)
  { schaums.dswj.Arrays.load(a,10,90);
    schaums.dswj.Arrays.print(a);
    shuffle(a);
    schaums.dswj.Arrays.print(a);
  }
  private static void shuffle(Object[] a)
  { Random random = new Random( );
    for (int i=0; i<a.length; i++)
      schaums.dswj.Arrays.swap(a,i,random.nextInt(a.length));
  }
}
```

2.16 测试 tally(String)方法。

```
import java.util.Random;
public class Pr0216
{ public static void main(String[] args)
  { String string="Welcome to the new millenium";
    System.out.println(string);
    int[] t = tally(string);
    for (int i=0; i<26; i++)
      System.out.println("Frequency of " + (char)('A'+i)
                         + " = " + t[i]);
  }
  private static int[] tally(String s)
  { int[] frequency = new int[26];
    for (int i=0; i<s.length( ); i++)
    { char ch = Character.toUpperCase(s.charAt(i));
      if (Character.isLetter(ch))
        ++frequency[(int)ch - (int)'A']; // count ch
    }
    return frequency;
  }
}
```

2.17 测试 innerProduct(double[], double[])方法。

```
public class Pr0217
```

标准分享网 www.bzfxw.com 免费下载

```
{ public static void main(String[] args)
  { double[] x = { 1.1, 2.2, 3.3, 4.4 };
    double[] y = { 2.0, 0.0, 1.0, -1.0};
    System.out.print("innerProduct(x,y)=" + innerProduct(x,y));
     }
  private static double innerProcuct(double[] x, double[] y)
  { double sum = 0.0;
    for (int i=0; i<x.length && i<y.length; i++)
          sum += x[i]*y[i];
    return sum;
  }
}
```

2.18 测试 outerProduct (double[], double[])方法。

```
public class Pr0218
{ public static void main(String[] args)
  { double[] x = { 1.1, 2.2, 3.3, 4.4 };
    double[] y = { 2.0, 0.0, -1.0 };
    double[][] z = outerProduct(x,y);
    for (int i=0; i<x.length; i++)
    { for (int j=0; j<y.length; j++)
        System.out.print("\t" + z[i][j]);
      System.out.println( );
    }
  }
  private static double [][] outerProduct(double[] x, double[] y)
  { double[][] z = new double[x.length][y.length];
    for (int i=0; i<x.length; i++)
      for (int j=0; j<y.length; j++)
        z[i][j] = x[i]*y[j];
    return z;
  }
}
```

2.19 测试 product(double[][], double[][])方法。

```
public class Pr0219
{ public static void main(String[] args)
  { double[][] x = { { 1.0, 2.0 },
                     { 3.0, 4.0 } };
    double[][] y = { { 20.0, -10.0 },
                     { 10.0, 20.0 } };
    double[][] z = product(x,y);
    for (int i=0; i<x.length; i++)
    { for (int j=0; j<y.length; j++)
```

```
        System.out.print("\t" + z[i][j]);
      System.out.println( );
    }
  }
  private static double[][] product(double[][] x, double[][] y)
  { double[][] z = new double[x.length][y[0].length];
    for (int i=0; i<x.length; i++)
      for (int j=0; j<y[0].length; j++)
      { double sum=0.0;
        for (int k=0; k<x[0].length; k++)
          sum += x[i][k]*y[k][j];
        z[i][j] = sum;
      }
    return z;
  }
}
```

2.20 测试 transpose(double[][])方法。

```
public class Pr0220
{ public static void main(String[] args)
  { double[][] x = { { 1.0, 2.0, 3.0 },
                     { 4.0, 5.0, 6.0 } };
    double[][] y = transpose(x);
    for (int i=0; i<y.length; i++)
    { for (int j=0; j<y[0].length; j++)
        System.out.print("\t" + y[i][j]);
      System.out.println( );
    }
  }
  private static double[][] transpose(double[][] x)
  { double[][] y = new double[x[0].length][x.length];
    for (int i=0; i<x[0].length; i++)
      for (int j=0; j<x.length; j++)
        y[i][j] = x[j][i];
    return y;
  }
}
```

2.21 测试帕斯卡三角形。

```
public class Pr0221
{ private static final int N=9;
  public static void main(String[] args)
  { int[][] p = pascal(N);
    for (int i=0; i<N; i++)
```

标准分享网 www.bzfxw.com 免费下载

```
    { for (int j=0; j<N; j++)
        System.out.print("\t" + p[i][j]);
      System.out.println( );
    }
  }

  private static int[][] pascal(int n)
  { int[][] p = new int[n][n];
    for (int j=0; j<n; j++)
      p[j][0] = p[j][j] = 1;
    for (int i=2; i<n; i++)
      for (int j=1; j<i; j++)
        p[i][j] = p[i-1][j-1] + p[i-1][j];
    return p;
  }
}
```

2.22 测试 Eratosthenes。

```
public class Pr0222
{ private static final int SIZE=1000;
  private static boolean[] sieve = new boolean[SIZE];
  public static void main(String[] args)
  { initializeSieve( );
    printSieve( );
  }
  private static void initializeSieve( )
  { for (int i=2; i<SIZE; i++)
      sieve[i] = true;
    for (int n=2; 2*n<SIZE; n++)
      if (sieve[n])
        for (int m=n; m*n<SIZE; m++)
          sieve[m*n] = false;
  }
  private static void printSieve( )
  { int n=0;
    for (int i=0; i<SIZE; i++)
      if (sieve[i]) System.out.print((n++%20==0?"\n":"\t")+i);
    System.out.println("\n" + n + " primes less than " + SIZE);
  }
}
```

2.23 用 java.util.Vector 实现 Eratosthenes 筛。

```
import java.util.Vector;
public class Pr0223
```

```
{ private static final int SIZE=1000;
  private static Vector sieve = new Vector(SIZE);
  public static void main(String[] args)
  { initializeSieve( );
    printSieve( );
  }
  private static void initializeSieve( )
  { sieve.add(Boolean.FALSE);
    sieve.add(Boolean.FALSE);
    for (int i=2; i<SIZE; i++)
      sieve.add(Boolean.TRUE);
    for (int n=2; 2*n<SIZE; n++)
      if (((Boolean)sieve.get(n)).booleanValue( ))
        for (int m=n; m*n<SIZE; m++)
          sieve.set(m*n,Boolean.FALSE);
  }
  private static void printSieve( )
  { int n=0;
    for (int i=0; i<SIZE; i++)
      if (((Boolean)sieve.get(i)).booleanValue( ))
        System.out.print((n++%10==0?"\n":"\t")+i);
    System.out.println("\n" + n + " primes less than " + SIZE);
  }
}
```

2.24 用 java.util.BitSet 实现 Eratosthenes 筛。

```
import java.util.BitSet;
public class Pr0224
{ private static final int SIZE=1000;
  private static BitSet sieve = new BitSet(SIZE);
  public static void main(String[] args)
  { initializeSieve( );
    printSieve( );
  }

  private static void initializeSieve( )
  { for (int i=2; i<SIZE; i++)
      sieve.set(i);
    for (int n=2; 2*n<SIZE; n++)
      if (sieve.get(n))
        for (int m=n; m*n<SIZE; m++)
          sieve.clear(m*n);
  }
```

标准分享网 www.bzfxw.com 免费下载

```
  private static void printSieve( )
  { int n=0;
    for (int i=0; i<SIZE; i++)
      if (sieve.get(i))
        System.out.print((n++%20==0?"\n":"\t")+i);
    System.out.println("\n" + n + " primes less than " + SIZE);
  }
}
```

2.25 Primes 类。

```
package schaums.dswj;
import java.util.*;
public class Primes
{ private static final int SIZE = 1000;
  private static int size = SIZE;
  private static BitSet sieve = new BitSet(size);
  private static int last = 1;

  static
  { for (int i=2; i<SIZE; i++)
      sieve.set(i);
    for (int n=2; 2*n<SIZE; n++)
      if (sieve.get(n))
        for (int m=n; m*n<SIZE; m++)
          sieve.clear(m*n);
  }

  private Primes( )
  {
  }

  public static void setLast(int n)
  { last = n;
  }

  public static void setLast( )
  { last = 1;
  }

  public static void setSize(int n)
  { size = n;
  }

  public static void setSize( )
```

```
  { size = 1000;
  }

  public static boolean isPrime(int n)
  { return sieve.get(n);
  }

  public static int next( )
  { while (++last<size)
      if (sieve.get(last)) return last;
    return -1;
  }

  public static void printPrimes( )
  { int n=0;
    for (int i=0; i<SIZE; i++)
      if (sieve.get(i))
        System.out.print((n++%10==0?"\n":"\t")+i);
    System.out.println("\n" + n + " primes less than " + SIZE);
  }
}
import schaums.dswj.Primes;
public class Pr0225
{ public static void main(String[] args)
  { Primes.printPrimes( );
    for (int n=1; n<=10; n++)
      System.out.println(n + ".\t" + Primes.next( ));
  }
}
```

2.26 测试素数因子方法。

```
import schaums.dswj.Primes;
import java.util.Random;
public class Pr0226
{ private static final int N=10;
  private static final int RANGE=2000;
  private static Random random = new Random( );

  public static void main(String[] args)
  { for (int i=0; i<N; i++)
    { int n = random.nextInt(RANGE);
      System.out.println(n + " = " + Primes.factor(n));
    }
  }
```

标准分享网 www.bzfxw.com 免费下载

```
}
```

在 Primes 类中加入以下方法：

```
public static String factor(int n)
  { String primes="";
    int p = next();
    while (n>1)
    { if (n%p==0)
      { primes += (primes.length()==0?"":"*") + p;
        n /= p;
      }
      else p = next();
      if (p == -1)
      { primes += " OVERFLOW";
        break;
      }
    }
    setLast();
    return primes;
  }
```

2.27 测试哥德巴赫猜想。

```
import schaums.dswj.Primes;
public class Pr0227
{ public static void main(String[] args)
  { final int N=10000;
    Primes.setSize(N);
    System.out.println("4 = 2+2");
    for (int n=6; n<100; n += 2)
    { System.out.print(n);
      for (int p=3; p<=n/2; p += 2)
        if (Primes.isPrime(p) && Primes.isPrime(n-p))
          System.out.print(" = "+p+"+"+(n-p));
      System.out.println( );
    }
  }
}
```

2.28 找出费马素数。

```
import schaums.dswj.Primes;
public class Pr0228
{ public static void main(String[] args)
  { final int N=10000;
    Primes.setSize(N);
```

```
    for (int p=0; p<5; p++)
    { int n = (int)Math.pow(2,Math.pow(2,p)) + 1;
      if (Primes.isPrime(n))
        System.out.println("p = "+p+", n = 2^2^p = "+n);
    }
  }
}
```

2.29 找出巴比奇素数。

```
import schaums.dswj.Primes;
public class Pr0229
{ public static void main(String[] args)
  { final int N=10000;
    Primes.setSize(N);
    for (int x=0; x<50; x++)
    { System.out.print(x);
      int n = x*x + x + 41;
      if (Primes.isPrime(n))
        System.out.println("\t"+n+" is prime");
      else System.out.println( );
    }
  }
}
```

2.30 找出孪生素数。

```
import schaums.dswj.Primes;
public class Pr0230
{ public static void main(String[] args)
  { final int N=1000;
    Primes.setSize(N);
    int n=Primes.next( );
    while (n<0.9*N)
    { if (Primes.isPrime(n+2))
        System.out.println(n + "\t" + (n+2));
      n = Primes.next( );
    }
  }
}
```

2.31 寻找平方数间的素数。

```
import schaums.dswj.Primes;
public class Pr0231
{ public static void main(String[] args)
  { final int N=10000;
```

标准分享网 www.bzfxw.com 免费下载

```
    Primes.setSize(N);
    for (int n=1; n<100; n++)
      for (int i=n*n+1; i<(n+1)*(n+1); i++)
        if (Primes.isPrime(i))
        { System.out.println(n*n + " < " + i + " < " + (n+1)*(n+1));
          break;
        }
  }
}
```

2.32 找出梅森尼素数。

```
import schaums.dswj.Primes;
public class Pr0232
{ public static void main(String[] args)
  { final int N=10000;
    Primes.setSize(N);
    for (int p = Primes.next( ); p<30; p = Primes.next( ))
    { int n = (int)Math.round(Math.pow(2,p)) - 1;
      System.out.print(p + "\t2^" + p + "-1 = " + n);
      if (Primes.isPrime(n)) System.out.println(" is prime ");
      else System.out.println(" is not prime ");
    }
  }
}
```

2.33 找出回文数。

```
import schaums.dswj.Primes;
public class Pr0233
{ public static void main(String[] args)
  { final int N=1000;
    Primes.setSize(N);
    for (int i=0; i<N; i++)
    { int p = Primes.next( );
      if (isPalindromic(p)) System.out.print(p+"\t");
    }
    System.out.println( );
  }

  private static boolean isPalindromic(int n)
  { if (n<0) return false;
    int p10=1;
    // make p10 is the greatest power of 10 that is < n
    while (p10<n)
      p10 *= 10;
```

```
    p10 /= 10;
    while (n>9)
    { if (n/p10 != n%10) return false;
      n /= 10;    // remove rightmost digit from n
      p10 /= 10;
      n %= p10;  // remove leftmost digit from n
    }
    return true;  // single digit integers are palindronic
  }
}
```

标准分享网 www.bzfxw.com 免费下载

第3章　Java高级特性

3.1　继承

面向对象程序设计的中心原理之一是类的继承（inheritance）。我们说类Y继承X，指Y通过使用X中的所有属性成员和方法成员，并定义自己的成员，从而扩展类X。在这种情况下，我们也称Y是X的子类。同样地，也可以把Y称为X的孩子类，X是Y的父类。从概念上讲，父子类关系所导致的结果是类Y中的实例比X中的实例更加具体，而X中的实例比Y中的实例更为一般。

例3.1　扩展Point类

下面将ColoredPoint定义成类Point的子类，Point类在例1.7中定义。

```
public class ColoredPoint extends Point
{ private String color="black";

  public ColoredPoint(double x, double y, String color)
  { super(x, y );  //调用父类的构造函数
    this.color = color;
  }

  public String getColor( )
  { return color;
  }

  public void setColor( String color)
  { this.color = color ;
  }

  public boolean equals( Object object)
  { if( object = = this ) return  true;
    if ( object,getClass( ) != this.getClass( ) return false;
    if ( object.hashCode( ) != this.hashCode( ) )  return false;
    ColoredPoint point = (ColoredPoint) object;
    Return( x == point.x && y == point.y && color == point.color);
  }

  public int hashCode( )
   {  return ( new Double(x) ).hashCode( ) + ( new Double ( y)).hashCode( )
```

```
                + color.hashCode( );
    }

    public String toString( )
    { return new String( " (" +x + "," + y +", " + color +" ) ") ;
}
}
```

从上可以看出，子类 ColoredPoint 定义了自己的构造器方法，一个新加的存取访问方法 getColor()和一个新的成员修改方法 setColor(String color) （也称 mutator 型方法）。并对父类 Point 中已有的方法 equals(Object)，hashCode()和 toString()进行了重新定义。

请注意为了使子类 ColoredPoint 中能访问到父类 Point 中的成员 x 和 y，成员 x 和 y 在类 Point 中应当声明为 protected，而不能是 private。

下面是一个测试驱动程序：

```
public class ExC301
{ public static void main(String[] args)
  { ColoredPoint p = new ColoredPoint(2,3,"green");
    System.out.println("p = " + p);
    ColoredPoint q = new ColoredPoint(2,3,"green");
    System.out.println("q = " + q);
    if (q == p) System.out.println("q == p");
    else System.out.println("q != p");
    if (q.equals(p)) System.out.println("q equals p");
    else System.out.println("q does not equal p");
    q.setColor("red");
    System.out.println("q = " + q);
    if (q.equals(p)) System.out.println("q equals p");
    else System.out.println("q does not equal p");
  }
}
```

该程序的输出为：

```
p = (2.0,3.0, green)
q = (2.0,3.0, green)
q != p
q equals p
q = (2.0,3.0, red)
q does not equal p
```

3.2 多态性

使继承成为如此有用的原因之一是对子类中的对象而言，当作为参数传递给一个方

标准分享网 www.bzfxw.com 免费下载

法时，可以当作父类的一个实例来看待，这种可使对象以多种形态出现的能力就称为多态性。

例 3.2　多态的使用

```
public class Pr0302
{ public static void main(String[] args)
  { ColoredPoint p = new ColoredPoint(2,3,"green");
    System.out.println("p = " + p);
    Point q = new Point(-2,0);
    System.out.println("q = " + q);
    System.out.println("distance(p,q) = " + distance(p,q));
  }
  private static double distance(Point p, Point q)
  { double dx = p.getX() - q.getX();
    double dy = p.getY() - q.getY();
    return Math.sqrt(dx*dx + dy*dy);
  }
}
```

该程序的输出为：

```
p = (2.0,3.0, green)
q = (-2.0,0.0)
distance(p,q) = 5.0
```

在上述程序中，对象 p 是类 ColoredPoint 的一个实例，在方法 distance(Point, Point)中允许 p 作为它父类 Point 的一个实例而成为第一个参数，因此，p 就以多种形态出现（同时作为类 ColoredPoint 和类 Point 的一个实例），也就是说，它是多态的。

所有的 Java 类，不管是标准库中的还是由程序员独立编写的，均存在于一个庞大的继承层次中，该层次结构是以 java.lang.Object 类为根的一个树形结构，其他类均位于该惟一的根类（父类）之下，因此，每个 Java 类均是类 Object 的直接或间接子类。

这里给出的树型类图（见图 3.1）仅包括 Java 1.2 标准库中 1462 个类中的 40 个，仅为 3%左右。例如，图 3.1 中表明 Integer 类是类 Number 的子类，而 Number 又是 Object 类的子类。

Java 类层次树同时也表明了多态是如何工作的，因为任何一个后代类的实例均可作为一个参数传递给方法调

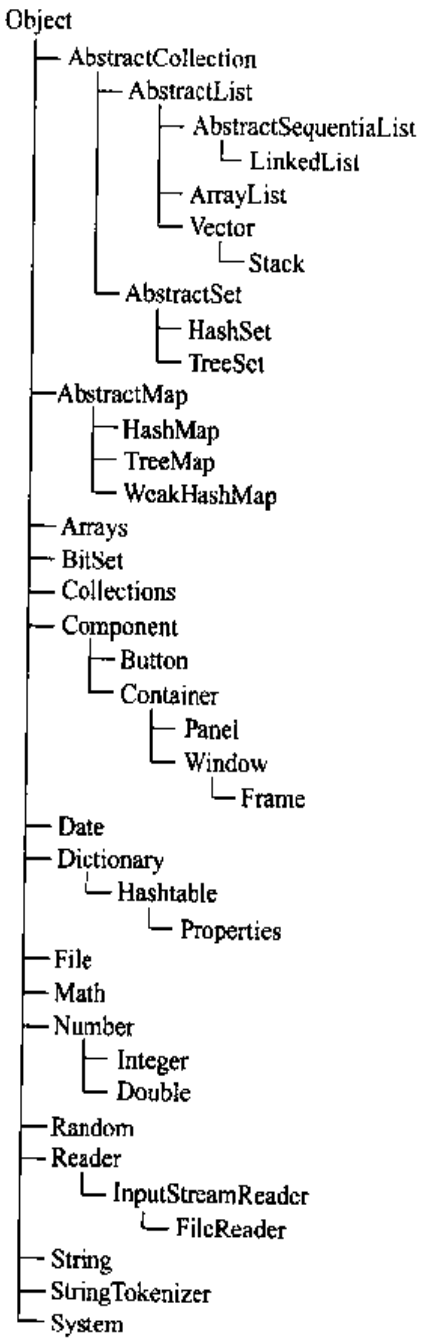


图 3.1　类层次结构图

用，只要该方法调用中对其参数的类型要求是该祖先类就行。我们可以简单地从叶结点往 Object 根方向一看就可知一个对象可以以怎样的形态出现，例如，一个 Frame 对象可以作为参数传递给那些要求类型为 Frame、Window、Container、Component 以及 Object 的方法调用。

正因为该层次结构，Java 中的多态意味着任何对象均可以传递给参数类型为 Object 的方法调用，任何对象均可调用 Object 类中所定义的任何方法（Object 中没有私有成员）。

3.3 类型转换

Java 中的每一个表达式均有一个类型，而类型可以由多种转换机制所转换。类 String 中的并置运算符+将把其他类型自动转换为 String 类型，如：

```
int n=44;
String s= "n= "+ n;
```

在表达式"n="+n 中，运算符+将把 int n 转换成 String 对象“44”，这种转换称为串转换。当非串类型是一个 class 类型时，此时将调用 class 类型的 toString()方法来完成转换工作，该方法可从 Object 类中继承。

在数字表达式中，当存在不同的数字类型时，窄类型值将自动转换为宽类型值，转换关系见图 3.2。例如：

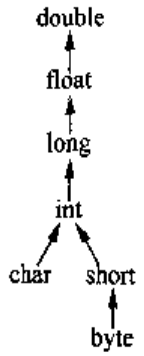


图 3.2 基本类型的宽型转换

```
double x= 3.14159 / 4;
```

表达式 3.14159 / 4 包括双精度值 3.141 59 和整型值 4，因此，在除法运算开始之前，整型值 4 将被提升为双精度类型，这也称为数字提升。

在赋值语句中，当一个窄类型赋值到一个宽类型时，也将完成相同的转换，例如：

```
double x= 44;
```

上述赋值语句将把整型值 44 赋给双精度型变量 x，在赋值开始之前，整型值将转换为相等的双精度型值，这种赋值也称为赋值转换。

赋值转换同样将发生在引用类型赋值操作上。

例 3.3 赋值转换

```
class X { int a; }
class Y extends X { int b; }
public class Ex0303
{ public static void main(String[] args)
  { X x = new X();
    System.out.println("x.getClass() = "+ x.getClass());
    Y y = new Y();
```

标准分享网 www.bzfxw.com 免费下载

```
        System.out.println("y.getClass() = "+ y.getClass());
        x = y;  // 将类 X 的引用赋值转换为类 Y 的引用
        System.out.println("x.getClass() = "+ x.getClass());
    }
}
```

该程序的输出为：

```
x.getClass() = class X
y.getClass() = class Y
x.getClass() = class Y
```

赋值语句 x = y 把一个窄类型赋给一个宽类型 x ，因而使得 x 转换为类型 y。

在 3.2 节中描述的多态也称为方法调用转换，即当参数传递给方法调用时一种类型转换为另一种类型，这种类型的转换也发生在基本类型身上，例如：

```
int n = 44;
double y= Math.sqrt(n ); //方法调用转换
```

这里，进行方法调用 Math.sqrt(n)时，整型值 44 将转换为 double 型值，这是因为所需参数类型为 double 的缘故。

最后，还可以显示使用类型转换子对类型进行强制转换。

例 3.4　强制转换引用类型

```
class X { public String toString() { return "I am an X."; } }
class Y extends X { public String toString() { return "I am a Y."; } }
public class Ex0304
{ public static void main(String[] args)
  { Y y = new Y();
    System.out.println("y: " - y);
    X x = y;
    System.out.println("x: " + x);
    Y yy = (Y)x;        // 将 X 强制转换成类 Y
    System.out.println("yy: " + yy);
    X xx = new X();
    System.out.println("xx: " + xx);
    yy = (Y)xx;       // 运行时错误，XX 不能强制转换成类 Y
    System.out.println("yy: " + yy);
  }
}
```

该程序的输出为：

```
y: I am a Y.
x: I am a Y.
```

```
yy: I am a Y.
xx: I am an X.
Exception in thread "main" java.lang.ClassCastException: X
        at Testing.main(Testing.java)
```

上述程序中，表达 yy= (Y) x 将把 x 强制转换为类型 Y，这种把值 x 解释为类型 Y 的转换之所以可行是由于 x 本身就来源于 y。在编译时强制转换 yy= (Y) xx 将把 xx 转换为类型 Y，但在运行时刻值 xx 不能解释为类型 Y 时，将发生 ClassCastException 异常。

例 3.4 中的强制转换展示了成功与不成功的引用类型转换，在下一例子程序中，将展示基本类型的强制窄型转换，包括成功与不成功的转换操作。基本类型的窄型转换关系见图 3.3。

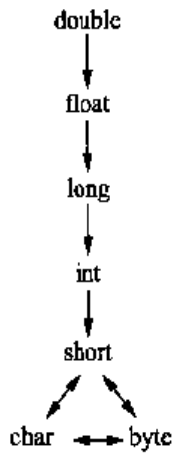


图 3.3 基本类型的窄型转换

例 3.5 强制基本类型转换

```
public class Ex0305
{ public static void main(String[] args)
  { double x = 44.0;
    System.out.println("x = " + x);
    float y = (float)x;    // narrowing cast from double to float
    System.out.println("y = " + y);
    int n = (int)y;        // narrowing cast from float to int
    System.out.println("n = " + n);
    short m = (short)n;    // narrowing cast from int to short
    System.out.println("m = " + m);
    n = 65536 + 4444;
    System.out.println("n = " + n);
    m = (short)n;          // narrowing cast from int to short
    System.out.println("m = " + m);
    y = (float)m;          // widening cast from short to float
    System.out.println("y = " + y);
    y = (float)n;          // widening cast from int to float
    System.out.println("y = " + y);
  }
}
```

该程序的输出是：

```
x = 44.0
y = 44.0
n = 44
m = 44
n = 69980
```

标准分享网_www.bzfxw.com 免费下载

```
m = 4444
y = 4444.0
y = 69980.0
```

程序中前 3 个窄型转换均是成功的，对转换中的窄类型而言，值 44 是合适的，但是，整型值 69 980（65 535 + 4 444）不能成功地转换为 short 型，可以从程序的输出中看到，为 m 赋给了一个错误的值 4 444。这是因为把 int 型值窄化为 short 时总获得该整型值被 65 536（216）所除时的余数。

同样，宽型强制转换 y= (float) m 将把一个 short 型整数成功转换为一个 float 型浮点数，但对 y = (float) n 而言，在把 int 型整数转换为 float 型浮点数的过程中将产生失败，这是因为整型值 69 980 对浮点值而言，有太多的有效数位，注意该值与把一个 int 型值在窄化成 short 型值所产生的数字值是一样的。

3.4　Object 类

类 Object 是 Java 中各类的最终祖先类，所有其他类均是它的后裔，Object 类中的每一个方法将为所有其他类继承，下面列出它的主要方法：

```
public class  Object
{ protected Object      clone();
  public  boolean       equals(Object );
  public final  Class   getClass();
  protected int         hashCode();
  public                Object();
  public   String       toString();
}
```

上述类中的大多数方法将会被子类覆盖 (overridden)。

方法 clone()返回对象的一个复制品，当子类没有覆盖该方法时，它仅简单地按成员一一对应的的方式进行复制。

例 3.6　不正确的克隆

```
class Widget implements Cloneable
{ int n;
  Widget w;
  Widget(int n) { this.n = n; }
  public Object clone() throws CloneNotSupportedException
  { return super.clone();
  }

  public String toString()
```

```
  { return "(" + n + "," + w.n + ")";
  }
}

public class Ex0306
{ public static void main(String[] args)
        throws CloneNotSupportedException
  { Widget x = new Widget(44);
    x.w = new Widget(66);
    System.out.println("x = " + x);
    Widget y = (Widget)x.clone();
    System.out.println("y = " + y);
    x.n = 55;
    x.w.n = 77;
    System.out.println("x = " + x);
    System.out.println("y = " + y);
  }
}
```

该程序的输出为：

```
x = (44,66)
y = (44,66)
x = (55,77)
y = (44,77)
```

在上述程序中，对象 x 中的 n 成员初始值为 44，然后 w 成员在第二行直接进行实例化，使其 x.w.n 的初值为 66。

对象 y 以 x 的克隆形式构造，方法 Object.clone()仅简单地对 x 的两个成员 n 和 w 进行复制，而不会对另一个类型为 y.w 的进行初始化赋值，因而 x.w 所引用的 widget 型对象没有被复制。

当 x.n 赋值为 55 时，它对 y.n 没有任何影响，它仍然保持它原有的数值 44。但当 x.w.n 赋值为 77 时，y.w.n 的值也变为 77，因此，y 与 x 就不是完全独立的了，这样，就使得克隆不完整，参见图 3.4。

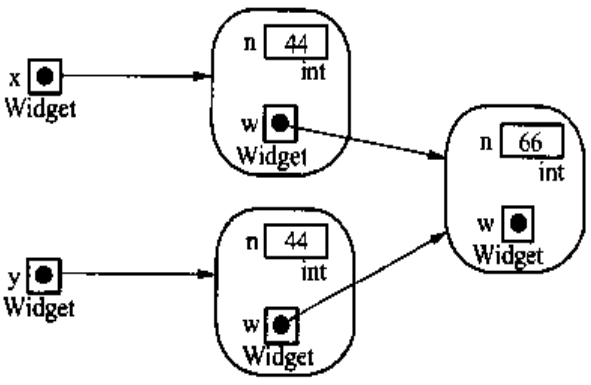


图 3.4 未克隆的引用对象

例 3.7 正确克隆

下面的程序对例 3.6 中的克隆方法进行了改写。

```
class Widget implements Cloneable
```

标准分享网 www.bzfxw.com 免费下载

```
{ int n;
  Widget w;
  Widget(int n) { this.n = n; }
  public Object clone() throws CloneNotSupportedException
  { Widget newWidget = new Widget(this.n);
    Widget y = newWidget;
    Widget t = this.w;
    while (t != null)
    { y.w = new Widget(t.n);
      y = y.w;
      t = t.w;
    }
    return newWidget;
  }

  public String toString()
  { return "(" + n + "," + w.n + ")";
  }
}

public class Ex0307
{ public static void main(String[] args)
       throws CloneNotSupportedException
  { Widget x = new Widget(44);
    x.w = new Widget(66);
    System.out.println("x = " + x);
    Widget y = (Widget)x.clone();
    System.out.println("y = " + y);
    x.n = 55;
    x.w.n = 77;
    System.out.println("x = " + x);
    System.out.println("y = " + y);
  }
}
```

该程序的输出为：

```
x = (44,66)
y = (44,66)
x = (55,77)
y = (44,66)
```

从输出结果可以看出，该程序中的克隆方法不仅对 widget 类中的成员进行复制，同样也对由该对象所串接的每一个 widget 型对象进行复制，并使克隆的对象完全独立于被克隆对象，使对其中一个的改变不会影响到另一个，参见图 3.5。

方法 equals()应当与对 clone()方法同样的方式进行覆盖，所有相应的串连成员都应当进行比较（见例 1.7 和例 3.4）。

方法 getClass()返回一个 Class 对象，它表示该实例对象所属的类对象，这在进行对象的相等性测试时是有用的（见例 1.7 和问题 3.4）。

方法 hashCode ()返回一个 int 型整型值，该整型值起到标识该对象的作用，对不同的对象而言，哈希代码可能是不一样的，对复杂对象而言，该值应当从各构成成员而产生（见例 1.7 和问题 3.5）。

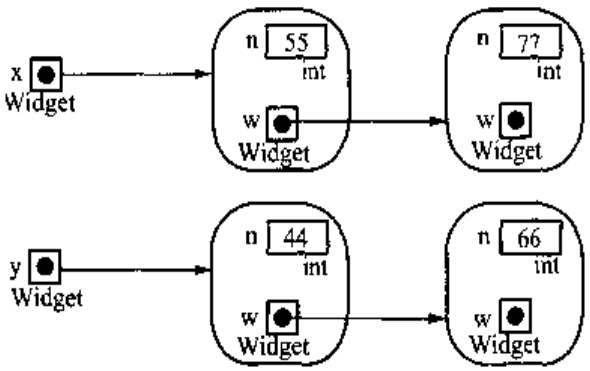


图 3.5 完全克隆的引用对象

方法 toString()返回一个 String 型对象，它表示对象的内容（见例 1.7 和例 3.6）。

3.5 抽象类

一个抽象类（abstract class）是至少包含一个抽象方法的类，一个抽象方法是指一个目前还没有实现的方法，仅有方法头声明。关键字“abstract”用于标识一个抽象类和抽象方法。

例 3.8 一个抽象的 Sequence 类

```
public abstract class Sequence
{ protected int length = 0;
  public int getLength( )
  { return length;
   }

  public abstract void append(Object object);
  // 将给定对象添加到序列的尾端

  public int count( Object object)
  {
   int c = 0;
   for( int i=0; i< getLength( ); i++)
if ( object.equals( get( i) )  ++c;
   return  c;
  }

  public abstract Object get( int index );
  // 返回给定索引位置的那个对象，如索引值超出了序列范围，则返回空值

  public int indexOf(Object objext)
  {
```

标准分享网 www.bzfxw.com 免费下载

```
   for( int i=0; i < length; i++)
if( object.equals( get( i) )) return i;
   return -1;
   }
  public abstract Object remove( int index );
  //返回给定索引位置的对象，并将该对象从序列中移出，当索引值超出序列的长度范围时，返回空值

  public abstract boolean remove(Object  object);
  // 将对象 Object 从序列中的第一次出现的地方移出，操作成功时，返回真值

  public abstarct Object set( int index, Object object);
   // 用对象 Object 替换指定索引位置的对象后，返回该对象，在操作不成功时，返回空值

  public String tostring( )
  { if (length == 0) return "( )";
String s = "(" + get( 0) ;
for ( int i=1; i<length; i++)
  s += "," + get(i );
return s + ")";
}
  }
```

从上可以看出，该类中有5个抽象方法，以及4个具体的非抽象方法，另外还有一个属性成员。请注意每个抽象方法的声明均以分号“;”结束。

定义抽象方法的目的在于允许某些方法以各种不同的方式加以实现，它们将在子类中加以完成。

例 3.9　类 Sequence 的子类 ArraySequence

```
public class ArraySequence extends Sequence
{ protected Object[] a;
  protected int capacity=16;
  // 不变式: a[i] != null for 0 <= i < length;

  public ArraySequence()
  { a = new Object[capacity];
  }

  public void append(Object object)
  { if (length==capacity) // 使序列的长度加倍
    { Object[] tmp = a;
      capacity *= 2;
      a = new Object[capacity];
      for (int i=0; i<length; i++)
```

```
        a[i] = tmp[i];
      }
      a[length++] = object;
    }

    public Object get(int index)
    { if (index >= length) return null;
      return a[index];
    }

    public Object remove(int index)
    { if (index >= length) return null;
      Object x = a[index];
      for (int i=index; i<length; i++)
        a[i] = a[i+1];
      --length;
      return x;
    }

    public boolean remove(Object object)
    { int i=indexOf(object);
      if (i == -1) return false;
      remove(i);
      return true;
    }

    public Object set(int index, Object object)
    { if (index >= length) return null;
      Object x = a[index];
      a[index] = object;
      return x;
    }
  }
```

该子类以Objects对象数组的形式实现sequence结构，提供一个缺省的构造器，并实现Sequence类中的所有5个抽象方法。

下面是一个测试驱动程序：

```
public class Ex0309
{ public static void main(String[] args)
  { ArraySequence s = new ArraySequence();
    System.out.println("s = " + s);
    s.append("Chile");
    s.append("China");
```

标准分享网 www.bzfxw.com 免费下载

```
        s.append("Congo");
        s.append("Egypt");
        s.append("China");
        s.append("India");
        s.append("Italy");
        s.append("China");
        System.out.println("s = " + s);
        System.out.println("s.getLength() = " + s.getLength());
        System.out.println("s.count(\"China\") = " + s.count("China"));
        System.out.println("s.get(5) = " + s.get(5));
        System.out.println("s.indexOf(\"China\") = "
                           + s.indexOf("China"));
        System.out.println("s.remove(5) = " + s.remove(5));
        System.out.println("s.remove(\"China\") = " + s.remove("China"));
        System.out.println("s.indexOf(\"China\") = "
                           + s.indexOf("China"));
        System.out.println("s.set(2,\"Japan\") = " + s.set(2,"Japan"));
        System.out.println("s = " + s);
        System.out.println("s.getLength() = " + s.getLength());
    }
}
```

该程序的输出是:

```
s = ()
s = (Chile,China,Congo,Egypt,China,India,Italy,China)
s.getLength() = 8
s.count("China") = 3
s.get(5) = India
s.indexOf("China") = 1
s.remove(5) = India
s.remove("China") = true
s.indexOf("China") = 3
s.set(2,"Japan") = Egypt
s = (Chile,Congo,Japan,China,Italy,China)
s.getLength() = 6
```

例3.9使用一个数组来实现定义在例3.8中的Sequence类，当然，还有Sequence类的其他实现。例如，我们可以使用一个类似于例3.6中Widget类所定义的链结构来加以实现，对采用何种实现方式的决定留以后讨论。

3.6 接口

一个抽象数据类型，或 ADT，是一种数据类型的行为描述，而不考虑它的实现细节。在 Java 语言中，一种数据类型就是一个类，而一个抽象数据类型即为一个接口。

一个接口是抽象类的一种特定形式，其中，没有任何方法实现，并且所有属性成员均为非 final 属性（non-final ）。它起到一个蓝图或协议作用，实现该接口的任何类的任何对象均可通过方法来完成接口中所描述的功能。因此，其他方法可以声明它的参数类型可以是某种接口类型，以及类类型和基本类型。编译器允许一个对象作为参数传递给该方法，只要该对象是实现该接口的类的一个实例，即使该类是一个抽象类。

例 3.10 一个 Sequence 接口

下面是一个接口定义，它可以代替例 3.8 中所定义的抽象类。

```
public interface Sequential
{ public abstract void append(Object object);
  // 将给定对象添加到序列的尾端
  public int count(Object object);
  // 返回给定对象在序列中的出现次数

  public abstract Object get(int index);
  // 返回给定索引位置的对象，当索引值超出序列范围时返回空值

  public int getLength();
  // 返回序列中的元素个数

  public int indexOf(Object object);
  // 返回给定对象在序列中第一次出现之前的元素的个数，当给定对象不在序列中时，返回-1

  public abstract Object remove(int index);
  // 返回并移去给定索引位置的对象，当索引值大于或等于序列长度时，返回空值

  public abstract boolean remove(Object object);
  // 将对象 object 从序列中的第一次出现的位置移出，当仅当操作成功时，返回值 true

  public abstract Object set(int index, Object object);
  // 用给定对象替换指定索引位置的对象后，返回该对象，当操作不成功时，返回空值

  public String toString();
  // 返回一个显示该序列内容的字符串
}
```

注意接口中没有包含任何这些代码，根据定义，它们只能包含方法声明和常量定义。

同时还应注意，为了与 Java 标准库的一致，我们使用名称作为类名而用形容词作为接口名，这将有助于我们记住一个类是一种类型，而一个接口只是一个行为的描述。例如，Sequential 接口说明了实现该接口的类的对象所应具有的行为，表现为一种“sequential”数据结构。

标准分享网 www.bzfxw.com 免费下载

可以使用类的方式来使用接口，如用于对变量以及对参量的声明，例如：

```
public static void print(Sequential sequence)
{  for( i=0; i<sequence.getLength(); i++)
     System.out.println(i + ":" + sequence.get(i) );
 }
```

对上述方法定义，即使没有 Sequential 接口的任何实现，仍可由编译器加以编译。

但为了使接口变得有用，接口必须加以实现，这可以通过定义一个类，该类通过实现子句（implements clause）来对接口中声明的方法加以实现。

例 3.11 一个实现 Sequential 接口的 ArraySequence 类

```
public class ArraySequence implements Sequential
{ protected int length = 0;
  protected Object[] a;
  protected int capacity=16;
  // INVARIANT: a[i] != null for 0 <= i < length;

  public ArraySequence()
  { a = new Object[capacity];
  }

  public void append(Object object)
  { if (length==capacity) // double the capacity
    { Object[] tmp = a;
      capacity *= 2;
      a = new Object[capacity];
      for (int i=0; i<length; i++)
        a[i] = tmp[i];
    }
    a[length++] = object;
  }

  public int count(Object object)
  { int c = 0;
    for ( int i=0; i<length; i++ )
      if ( object.equals(get(i)) ) ++c;
    return c;
}

  public Object get(int index)
  { if (index >= length) return null;
    return a[index];
```

```
}

public int getLength()
{ return length;
 }

public int indexOf(Object object)
{ for ( int i=0; i<length; i++ )
  if( object.equals(get(i))) return i;
return -1;
 }

public Object remove(int index)
{ if (index >= length) return null;
  Object x = a[index];
  for (int i=index; i<length; i++)
    a[i] = a[i+1];
  --length;
  return x;
}
public boolean remove(Object object)
{ int i=indexOf(object);
  if (i == -1) return false;
  remove(i);
  return true;
}

public Object set(int index, Object object)
{ if (index >= length) return null;
  Object x = a[index];
  a[index] = object;
  return x;
}

 public String toString( )
 {  if(length --0) return "( )";
    String s = "(" + get( 0) ;
for ( int i=0; i<length; i++ )
s += "," + get(i);
return s + ")";
   }
}
```

注意例 3.11 中使用了与例 3.8 和例 3.9 中相同的代码，这表明使用抽象类而不是接口的

标准分享网 www.bzfxw.com 免费下载

主要区别在于管理上的不同，当用接口时，所有执行代码必须位于其实现中，而用抽象类时，可以使部分执行代码位于抽象类中，而其余代码位于子类中。

在接口中不能包含任何执行代码方面，接口没有抽象类灵活，但从一个类可以实现多个接口来讲，接口比抽象类更灵活。Java 不支持多重继承，即不支持一个类同时扩展多个类。

3.7　包

一个包是一组类和接口的集合，在一个包中，每个类或接口名必须惟一，但不同包中的不同类和接口可以同名。例如，在 Java 标准库中，有两个不同的类取名为 Object，一个位于 java.lang 包中，而另一个位于 java.org.omg.CORBA 包中。从技术上讲，这两个名字是不同的，因为它们的完全名 java.lang.Object 和 java.org.omg.CORBA.Object 是不同的，换言之，类所属的包名实际上是类名本身的前缀。

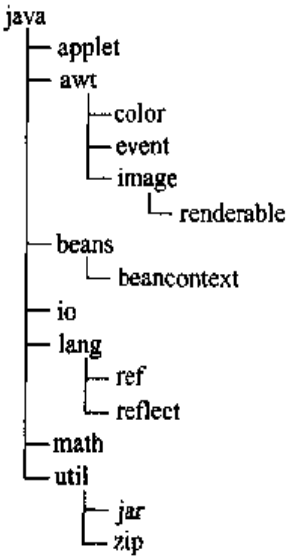


图 3.6　包间的层次结构

包组织成树形结构，正如类、接口和文件目录所组成的树形结构一样。事实上，Java 程序员通常把每一个包映射成一个惟一的文件目录，使包层次名与文件层次名相匹配。例如，包 java.awt.event 中的所有类将放到目录 java/awt/event/下（在 Windows 环境下，目录名为 java\awt\event\）。Java 包中的一些子包所组成的层次结构如图 3.6 所示。

多数 Java 开发环境（见附录 C）都支持包的使用，它们将包名中的段映射为包所在存储位置的子目录。例如，在 Windows 环境中，包名 books.schaums.dswj.util 将转换为 books\schaums\dswj\util.java。

3.8　异常处理

一个异常是一个运行时错误，将使程序的执行从异常条件产生源转移到异常处理，异常条件产生源也称为异常的引发者（thrower），而异常处理器也称捕获器。

通常，运行时错误被封装成 Throwable 类和其子类的一个实例，该类的直接子类是 Error 类和 Exception 类，继承层次根据各异常产生的原因进行了进一步细分，如图 3.7 所示。

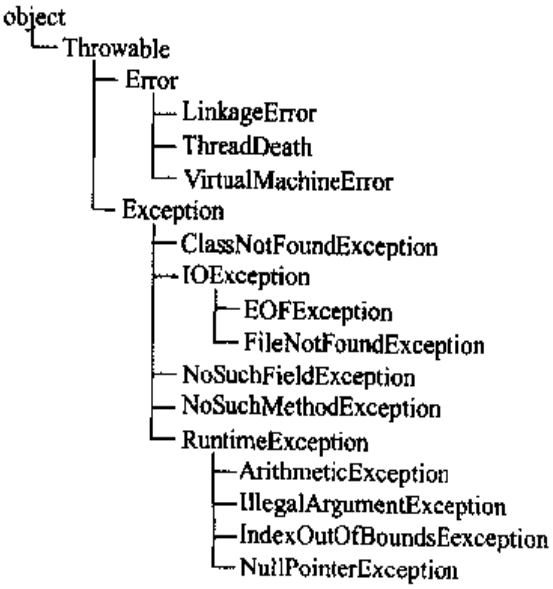


图 3.7　异常层次结构

下面的两个例子展示了处理异常的两种方法：通过 try .. catch 语句来捕获异常，或简单地在当前方法中加上 throw 语句将异常传递给该方法的调用者。

例 3.12 一个捕获异常的例子

```
public class Ex0312
{ public static void main(String[] args)
  { try
    { int n = Integer.parseInt("Abacadabra!");
      System.out.println("n = " + n);
    }
    catch (Exception e)
    { System.out.println("e = " + e);
    }
  }
}
```

```
e = java.lang.NumberFormatException: Abacadabra!
```

调用 Integer.parseInt(" Abacadabra!")将导致失败，这是由于传递给方法 Integer.parseInt() 的参数是字符串而不是整数的原因，该方法抛出一个 NumberFormatException 对象，该对象由 main()方法所捕获。Catach 块中的执行语句在异常被捕获后执行。

例 3.13 一个不被捕获的异常

```
public class Ex0313
{ public static void main(String[] args)
  { System.out.println("Try this...");
    int n = Integer.parseInt("Abacadabra!");
    System.out.println("n = " + n);
  }
}
```

```
Try this...
Exception in thread "main" java.lang.NumberFormatException:
Abacadabra!
        at java.lang.Integer.parseInt(Integer.java:409)
        at java.lang.Integer.parseInt(Integer.java:458)
        at Testing.main(Testing.java:9)
```

本程序中的方法调用与例 3.12 中的相同，但不在一个 try 块中执行，因此，在异常被抛出后，程序员失去了对异常的控制，换言之，程序的执行将崩溃，并打印出错误信息。

NumberFormatException 类是 RuntimeException 类的后裔，只有类 RuntimeException 的

标准分享网 www.bzfxw.com 免费下载

扩展类才能被处理，正如例 3.13 所示，而其他的异常（这是大量的）必须捕获，可由产生错误的方法捕获，也可由某些先前调用的方法进一步返回到调用栈。

必须对异常进行捕获（对任何不是 RuntimeException 类的扩展的类），因此，它们也称为被检查异常（checked exception）。

复　习　题

3.1　什么是继承？
3.2　什么是多态？
3.3　什么是强制类型转换？
3.4　什么是数字提升？
3.5　在 Java 中为什么称 Object 类是所有其他类的母类（也称父类）？
3.6　什么是抽象方法？
3.7　什么是抽象类？
3.8　什么是抽象类和接口的差异？
3.9　什么是子类和类扩展的差异？
3.10　什么是接口和抽象类的差异？
3.11　什么是抽象类和具体类的差异？
3.12　什么是扩展另一个类的类与实现某个接口的类间的差异？
3.13　什么是一个 try 块？
3.14　什么是被检查异常与不检查异常的区别？

习　　题

3.1　在 Point 类（见例 1.7）中增加第三维将其扩展为一个 Point3D 类，覆盖以下方法：

```
public boolean equals(Object  object);
public int hashCode( );
public String toString();
```

并增加以下方法：

```
private double z;
public Point3D(double x, double y);
public Point3D(double x, double y, double z);
public Point3D(Point3D q);
public double getZ();
public void setLocation(double x, double y, double z);
public void translate(double dx, double dy, double dz);
```

3.2 修改习题 3.1 中的 Point3D 类以反映前面的习题中对 Point 类所做的增加，覆盖以下方法：

```
public double magnitude();
public void expand(double dr);
```

并增加以下方法：

```
public static final Point3D ORIGIN;
public Point3D( );
public double distance( Point3D point);
```

3.3 请先预测下面程序的输出，然后运行该程序并与预测的输出进行比较。

```
class X
{ protected int a;
  protected int b = 22;
  public X( )
  {
   System.out.print("X ( ) :" + this );
   System.out.print(" a = " + a + ", b = " + b);
   a = 33;
   b =44;
   System.out.println(" a = " + a + ", b = " + b);
}
   public String toString( ){return " I am an X. ";}
   }
class Y extends X
{
  protected int c = 55;
  public Y ()
  {
System.out.print( "Y ( ) : " + this );
System.out.print(" a = " + a + ", b = " + b + ", c = " + c );
a = 66;
b = 77;
c = 88;
System.out.println(" a = " + a + ", b = " + b + ", c = " + c );
}
   public String toString( ) {return " I am a Y. ";}
  }
 public class Testing
{
 public static void main(String [ ] args)
  { Y y = new Y ( );
   }
```

标准分享网 www.bzfxw.com 免费下载

```
}
```

3.4 为例 3.6 中的 Widget 类实现 equals(Object) 方法。

3.5 为例 3.6 中的 Widget 类实现 hashCode() 方法。

复习题答案

3.1 继承描述了类和它的扩展类间的关系，扩展包括增加新的字段、新的方法以及对原类中方法的重新定义（覆盖）。把原有类称为父类或超类，而扩展类称为孩子类或子类。

3.2 多态指一个对象可以以另一个对象的形式来显示其行为，该外表对象与原对象实质上是一个对象，只是把它当作其父类或祖先类的一个成员。

3.3 类型强制转换是一种类型的值当作另一种类型来使用。对象不能改变它们的类型，但它们的值可以赋予另一种类型的对象，只要这两种类型位于继承树上从根到叶的同一条路径上。例如：

```
String string1 = "ABCDE";
Object object = (Object) string1;   // 将一个 string 型对象强制转换成一个 Object
                                    // 类的对象
String string2 = (String) object;   // 将一个 object 类的对象强制转换成一个 sring
                                    // 型对象
```

3.4 数值升迁类似于类型强制转换，它应用于基本类型。数字类型的值可以赋予更高数字类型的变量。例如：

```
int n = 44;
float x = n;     // 将一个整型数升迁为一个浮点数
double y = x;    // 将一个浮点数升迁为一个双精度数
```

3.5 在 Java 中 Object 类是惟一没有父类的类，它是 Java 继承树的根。

3.6 一个抽象方法只是一个简单的函数声明：包含方法修饰符，返回类型及其基调（signature）。

3.7 一个抽象类是至少有一个抽象方法的类。

3.8 一个抽象类可以有字段和一些已实现的方法，一个接口完全没有实现，只要常量和方法声明。

3.9 一个子类和类扩展间没有什么区别，它们是同义词。

3.10 一个接口是没有任何实现的类，也就是说，只是一系列的方法基调（signature），没有任何可执行的语句和字段定义。抽象类除了有一些实现方法和某些字段定义外与接口相同。抽象类中没有实现的方法在声明时应当用 abstract 关键字限定。

3.11 一个抽象类是一个具有一个或多个抽象方法（没有被实现）的类，而一个具体类中的所有方法均已被实现。

3.12 当一个类扩展另一个类时，它继承父类的所有方法（覆盖的除外）。当一个类是实现

一个接口时，它应当实现接口中所有的方法声明。

3.13 一个 try 块是 try … catch 语句中紧跟在 try 关键字之后的语句块，这是引发检查异常的方法所在的语句块，并且应当进行异常处理，除非方法本身将异常返回到调用它的更外层方法进行处理。

3.14 一个检查异常是一个必须捕获的异常，可由方法所在位置进行或由调用树中更外层的方法进行处理。引发检查异常的方法只能在 try 块中调用或由抛出异常的其他方法所调用。惟一不检查的异常是由 RunTimeException 类的后代类所引发的异常。

习 题 解 答

3.1 扩展 Point 类。

```
public class Point3D extends Point
{ protected double z;
  public Point3D(double x, double y)
  { this.x = x;
    this.y = y;
    this.z = 0;
  }
  public Point3D(double x, double y, double z)
  { this.x = x;
    this.y = y;
    this.z = z;
  }
  public Point3D(Point3D q)
  { this.x = q.x;
    this.y = q.y;
    this.z = q.z;
  }
  public double getZ()
  { return z;
  }
  public void setLocation(double x, double y, double z)
  { this.x = x;
    this.y = y;
    this.z = z;
  }
  public void translate(double dx, double dy, double dz)
  { x += dx;
    y += dy;
    z += dz;
  }
```

标准分享网 www.bzfxw.com 免费下载

```
  public boolean equals(Object object)
  { if (object == this) return true;
    if (object.getClass() != this.getClass()) return false;
    if (object.hashCode() != this.hashCode()) return false;
    Point3D point = (Point3D)object;
    return (x == point.x && y == point.y && z == point.z);
  }
  public int hashCode()
  { return (new Double(x)).hashCode() + (new Double(y)).hashCode()
                                + (new Double(z)).hashCode();
  }
  public String toString()
  { return new String("(" + (float)x + "," + (float)y
                      + "," + (float)z + ")");
  }
}
```

3.2　对 Point3D 类增加方法。

```
public static final Point3D ORIGIN = new Point3D();
public Point3D()
{ this.x = 0;
  this.y = 0;
  this.z = 0;
}
public double distance(Point3D point)
{ double dx = this.x - point.x;
  double dy = this.y - point.y;
  double dz = this.z - point.z;
  return Math.sqrt(dx*dx+dy*dy+dz*dz);
}
public double magnitude()
{ return distance(ORIGIN);
}
public void expand(double dr)
{ x *= dr;
  y *= dr;
  z *= dr;
}
```

3.3　程序执行的次序为：

1. 将各字段设置成缺省的初始值。
2. 调用 Y()构造方法。
3. 调用 X()构造方法。
4. 调用 Objext 构造方法。

5. 对X中的各字段进行初始化。
6. 执行X()构造方法。
7. 对Y中的各字段进行初始化。
8. 执行Y()构造方法。

因此程序所输出为：

```
X( ) : I am a Y. a = 0, b = 22 a = 33, b = 44
Y( ) : I am a Y. a = 33 , b = 44 , c = 55 a = 66, b = 77, c = 88
```

3.4 用于Widget类的equals(Object)方法。

```
public boolean equals(Object object)
{ if (object == this) return true;
  if (object.getClass() != this.getClass()) return false;
  Widget y = (Widget)object;
  Widget t = this;
  while (y != null || t != null)
  { if (y == null || t == null) return false;
    if (y.n != t.n) return false;
    y = y.w;
    t = t.w;
  }
  return true;
}
```

3.5 用于Widget类的hashCode()方法。

```
public int hashCode()
{ int hc = (new Integer(n)).hashCode();
  Widget tw = this.w;
  while (tw != null)
  { hc += (new Integer(tw.n)).hashCode();
    tw = tw.w;
  }
  return hc;
}
```

标准分享网 www.bzfxw.com 免费下载

第4章 递 归

递归函数是指调用自己的函数，这个有力的技术手段可以在不用 while 循环、do .. while 或 for 循环的情况下产生重复过程，因此可从一小段程序代码产生许多重要结果。

递归可为一个复杂问题获得一个非常优美的解，但它也可能被误用，写出非常低效率的代码。

递归代码通常来源于递归算法。

例 4.1 阶乘函数

阶乘函数在数学上定义为：

$$n!=\begin{cases}1, & if\ \ n=0\\ n(n-1)!, & if\ \ n>0\end{cases}$$

这是一个递归定义，因为阶乘又在等式的右边出现，该函数是由该函数自己定义的。

阶乘函数的前 10 个值展示在表 4.1 中，第一个值 0!由定义中的上半部分定义，0! = 1（对 n=0），其余的值由定义中的下半部分定义。

表 4.1 阶乘值表

n	$n!$	n	$n!$
0	1	5	120
1	1	6	720
2	2	7	5 040
3	6	8	40 310
4	24	9	362 880

对 $n=1$，$1!=n!=n\,(n-1)!=1(1-1)!=1(0)!=1(1)=1$。

对 $n=2$，$2!=n!=n\,(n-1)!=2(2-1)!=2(1)!=2(1)=2$。

对 $n=3$，$3!=n!=n\,(n-1)!=3(3-1)!=3(2)!=3(2)=6$。

对 $n=4$，$4!=n!=n\,(n-1)!=4(4-1)!=4(3)!=4(6)=24$。

对 $n=5$，$5!=n!=n\,(n-1)!=5(5-1)!=5(4)!=5(24)=120$。

注意该函数的增长速度。

例 4.2 阶乘函数的递归实现

当一个函数是用递归方式定义时，它的实现通常是将其直接翻译成递归定义，阶乘函数递归定义中的两部分可以直接翻译成两个 Java 语句：

```
public static int f (int n )
{  if ( n = = 0) return 1;  //basis
   return  n * f( n-1);   // recursive part

 }
```

下面是一个简单的用于阶乘方法的测试驱动程序：

```
public static void main( String[ ] srgs )
{  for ( int n = 0 ; n < 10; n ++ )
       System.out.println( "f (" + n + ") = " + f( n) );
}
```

该程序的输出应当如表 4.1 所示。

例 4.3 阶乘函数的迭代实现

阶乘函数也是易于用迭代方式来加以实现的：

```
public static int f ( int n )
{ int f =1;
   for ( int i=2; i<= n; i++)
      f *=i;
   return f ;
}
```

注意函数头与例 4.2 中的相同，只是函数体不一样，因此，允许对这两个不同的实现使用同样的测试程序，它们的输出应当是相同的。

4.1 递归基和递归体

每个递归函数必须要有一个递归基和递归体部分才能正确工作，递归基是用于终止递归的条件部分，而递归体部分是函数调用它自己的部分。

例 4.4 阶乘函数的递归基和递归体

在例 4.2 实现阶乘函数的 Java 方法中，递归基和递归体部分均用注释进行了标注，递归体部分用一个更小的值去调用方法。如开始用正整数 5，则随后的调用中分别用 4，3，2，1 以及 0。当遇到调用测试值为 0 时，递归基将被执行，因而终止进一步递归，并开始返回链的处理，依次返回 1，1，2，6，24，并最后获得返回值 120。

例 4.5 斐波那契数列

斐波那契数列为 1，1，2，3，5，8，13，21，34，55，…，从第三个开始，随后的每一个数均是其前两个数之和。这是一个极自然的递归定义函数：

标准分享网 www.bzfxw.com 免费下载

$$F_n=\begin{cases}0, & if\ \ n=0\\ 1, & if\ \ n=1\\ F_{n-1}+F_{n-2}, & if\ \ n>1\end{cases}$$

斐波那契数列的前 15 个值列在表 4.2 中，前两个值，F_0 和 F_1，均由定义中的前两部分确定：$F_0=0$（此时 $n=0$），$F_1=1$（此时 $n=1$），前两部分构成了递归基，所有其他值由定义中的递归体部分确定：

对 $n=2$，$F_2=F_n=F_{n-1}+F_{n-2}=F_{(2)-1}+F_{(2)-2}=F_1+F_0=1+0=1$

对 $n=3$，$F_3=F_n=F_{n-1}+F_{n-2}=F_{(3)-1}+F_{(3)-2}=F_2+F_1=1+1=2$

对 $n=4$，$F_4=F_n=F_{n-1}+F_{n-2}=F_{(4)-1}+F_{(4)-2}=F_3+F_2=2+1=3$

对 $n=5$，$F_5=F_n=F_{n-1}+F_{n-2}=F_{(5)-1}+F_{(5)-2}=F_4+F_3=3+2=5$

对 $n=6$，$F_6=F_n=F_{n-1}+F_{n-2}=F_{(6)-1}+F_{(6)-2}=F_5+F_4=5+3=8$

对 $n=7$，$F_7=F_n=F_{n-1}+F_{n-2}=F_{(7)-1}+F_{(7)-2}=F_6+F_5=8+5=13$

表 4.2 斐波那契数列

n	F_n	n	F_n
0	0	8	21
1	1	9	34
2	1	10	55
3	2	11	89
4	3	12	144
5	5	13	233
6	8	14	377
7	13		

例 4.6 斐波那契数列的递归生成

```
public static int fib(int n)
  { if (n < 2) return n;              // basis
    return fib(n-1) + fib(n-2);       // recursive part
  }
```

下面是测试斐波那契数方法的简单驱动程序：

```
  public static void main(String[] args)
  { for (int n=0; n<16; n++)
      System.out.println("fib("+n+") = "+fib(n));
}
```

上述程序的输出值应当与例 4.5 中给出的表 4.2 一致。

4.2 跟踪递归调用

对方法的执行过程进行跟踪通常有助于我们更清楚地理解它。

例 4.7 跟踪阶乘递归函数

图 4.1 所示是对定义在例 4.2 中的递归阶乘函数进行 f (5)调用时所进行的跟踪：

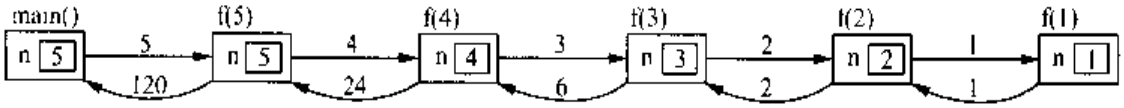


图 4.1 阶乘函数的递归调用关系

该调用是由 main()函数引发的，并把参数 5 传递给 f ()函数，因此，参数 n 的值为 5 ，f (5) 又把参数 4 传递给函数 f ()，它调用 f (4)。随后 f(4) 又调用 f(3)，将参数 3 传递给分 f()函数，这个过程（以递归的方式）直到调用 f (1)为止。由于调用参数为 1， 它立即返回 1，到达递归结束点。然后，f (2) 将 2×1=2 返回给 f (3)，继而 f (3) 将 3×2=6 返回给调用 f (4)，而 f (4) 又以 4×6=24 返回给调用 f (5)，最后，调用 f (5)将值 120 返回给 main()。

例 4.7 中的跟踪过程表明阶乘函数 f (n)的递归实现将产生 n –1 次递归调用，这与例 4.3 中的迭代实现相比是明显低效的。

例 4.8 斐波那契递归函数的跟踪

斐波那契函数（例 4.6）与阶乘函数相比（例 4.2）具有更多的递归，它包含了两个递归调用，这可从调用 fib (5)的跟踪中看出，跟踪过程见图 4.2。对斐波那契递归函数的调用源于 main () 函数，将参数 5 传递给 fib () 函数，此时，参数 n 的值为 5，因而它调用 fib (4) 和 fib (3)，分别传递参数 4 和 3。每个调用又产生两个递归调用，一直到递归基调用 f(1) 和 f (0)。这两个基调用均返回 1。递归调用依次将下级递归调用的返回值的和返回，并最终将结果值 8 返回给 main ()。

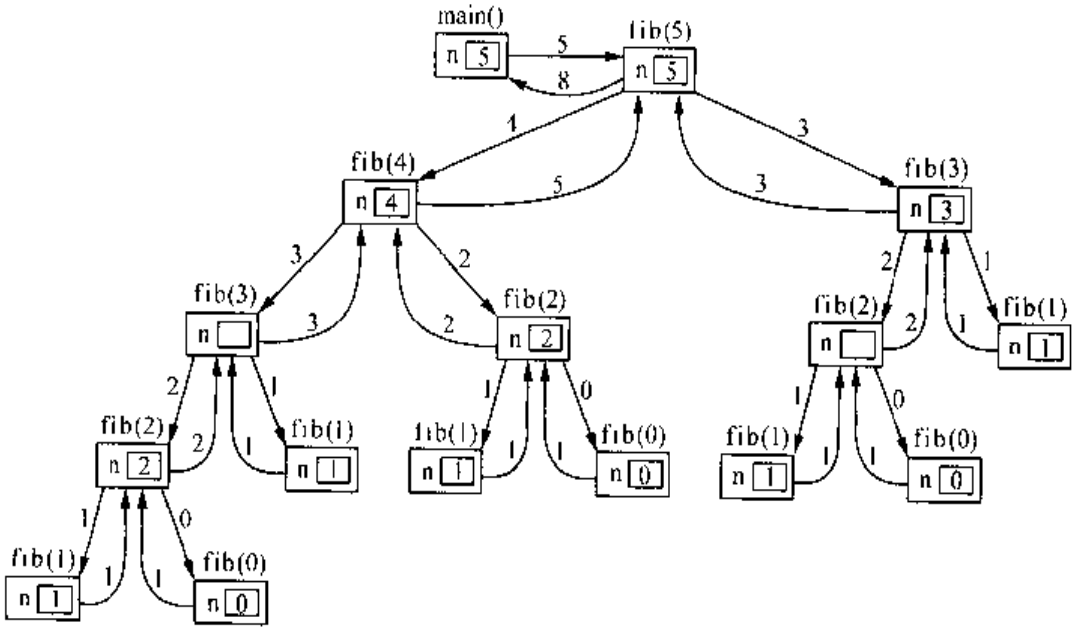


图 4.2 斐波那契函数的递归调用关系

标准分享网 www.bzfxw.com 免费下载

4.3 递归折半查找算法

在2.5节已经给出了非递归的折半查找算法，它使用分治（divide-and-conquer）策略，每次将序列分成两半，并在随后的一半中继续查找，这是一个自然的递归过程。

算法4.1 递归折半查找算法

（前置条件： $s = \{ s_0, s_1, \ldots, s_{n-1}\}$是n个有序数值所组成的排序序列，这些数值具有相同的类型 x。）

（后置条件：当 $s_i = x$ 时，返回索引值i，否则返回-1。）

1. 当序列为空时，返回-1。
2. 令 s_i 为序列的中间元素。
3. 如果 $s_i = x$ 时，返回索引值i。
4. 如果 $s_i < x$ 时，应用递归折半查找算法对位于 s_i 右边的子序列进行查找。
5. 应用递归折半查找算法对位于 s_i 左边的子序列 ss 进行查找。

注意到算法4.1中的前提条件要求查找序列事先排序。

例4.9 递归折半查找

```
public static int search( int [ ] a, int lo, int hi, int x)
{ // 前置条件: a[ 0] <= a [ 1 ] <= … <= a[ a.length -1 ];
  // 后置条件: 当a [ i ] = x并且i >=0时，返回索引值i，否则返回 1。)
  if ( lo > hi ) return -1;                                  // 步骤 1
  int i = (lo + hi) /2;                                      // 步骤 2
  if ( a [i] == x ) return i;                                   // 步骤 3
  else if ( a[i] < x) return search ( a, i+1, hi, x);        // 步骤 4
  else return  search( a, lo , i -1 , x);                    // 步骤 5
  }
}
```

定理4.1 递归折半查找算法的运行时间为 $O(\lg n)$

证明：这里的讨论与定理2.4中的基本相同，运行时间与递归调用次数成正比，每次调用处理一个子序列，该序列的长度是上个递归函数中处理的序列的长度的一半。因此，递归调用的数目与n能被连续分成两半的次数是一样的，该次数显然为 lgn 次。

例4.10 测试递归折半查找

```
import schaums.dswj.Arrays;
public class Ex0410
```

```
{ private static final int SIZE = 16;
  private static final int START = 40;
  private static final int RANGE = 20;
  private static int[] a = new int[SIZE];

  public static void main(String[] args)
  { Arrays.load(a,START,RANGE);
    java.util.Arrays.sort(a);
    Arrays.print(a);
    test();
    test();
    test();
    test();
  }

  public static void test()
  { int x = Arrays.load(START,RANGE);
    System.out.print("Searching for x = " + x + ":\t");
    int i = search(a,x);
    if (i >= 0) System.out.println("a[" + i + "] = " + a[i]);
    else System.out.println("i = " + i + " ==> x not found");
  }

  public static int search(int[] a, int x)
  { return search(a,0,a.length-1,x);
  }

  public static int search(int[] a, int lo, int hi, int x)
  { if (lo>hi) return -1;  // basis
    int i = (lo+hi)/2;
    if (a[i] == x) return i;
    else if (a[i] < x) return search(a,i+1,hi,x);
    else return search(a,lo,i-1,x);
  }
}
```

该程序的输出为：

```
   0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15
{ 45   45  45  45  48  48  49  50  51  52  53  54  54  58  58  59  }
Searching for x = 46;   i = -1 ==> x not found
Searching for x = 48;   a[ 5 ] = 48
Searching for x = 54;   a[ 11 ] = 54
Searching for x = 57;   i = -1 ==> x not found
```

上述测试与例 2.8 中的相似。

标准分享网 www.bzfxw.com 免费下载

注意，为了使 search(int [], int)函数的声明仅有两个参量，我们不得不将它进行包装，使得非递归的测试驱动程序能按先前形式进行调用。

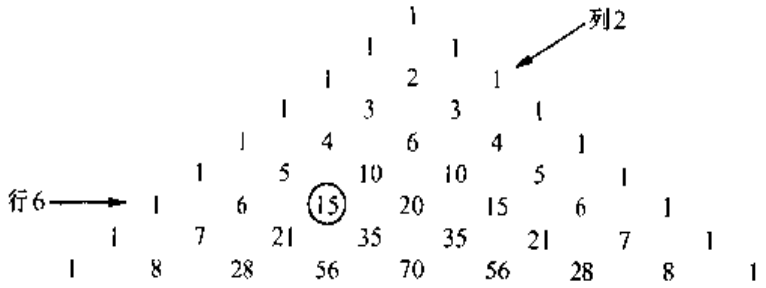
4.4 二项式系数

二项式系数是形如$(x+1)^n$的二项式表达式的展开式中的系数，例如：

$$(x+1)^6 = x^6 + 6x^5 + 15x^4 + 20x^3 + 15x^2 + 6x + 1$$

这里产生的 7 个系数分别为 1，6，15，20，15，6 和 1。

法国数学家帕斯卡(1623~1662)发现了二项式系数间的递归关系，将这些系数排列成一个三角形，他发现三角形中的每一个内部数为其直接上邻的两数之和。



例如，15 = 5 + 10。

令 $c(n,k)$表示所在行为 n ，列为 k（从 0 开始算起）的那个系数，例如，$c(6,2)=15$。这样，帕斯卡所发现的系数间的递归关系可表述为：

$$c(n,k) = c(n-1,k-1) + c(n-1,k) \quad \text{当 } 0 < k < n$$

例如，当 $n=6$，$k=2$ 时，$c(6,2) = c(5,1) + c(5,2)$。

例 4.11 二项式系数函数的递归实现

```
public static int c ( int n, int k )
{ if ( k == 0 | | k == n ) return 1;        //递归基
  return c ( n -1 , k -1 ) + c ( n -1 , k ) ;   //递归体部分
}
```

递归基包括了三角形中的最左边和最右边的系数值，此时 k=0 或 k =n。

二项式系数与组合数学中的组合数是一样的，它可由下面的公式计算得到：

$$c(n,k) = \frac{n!}{k!(n-k)!} = \left(\frac{n}{1}\right)\left(\frac{n-1}{2}\right)\left(\frac{n-2}{3}\right)\cdots\left(\frac{n-k+1}{k}\right)$$

在组合数学中，组合数通常写为：

$$c(n,k)=\binom{n}{k}$$

读作从 n 中选 k 个的组合数。例如，从 8 中选 3 个的组合数为 c (8, 3)= (8/1) (7 /2) (6/3)=56。

例 4.12 二项式系数函数的迭代实现

下面的程序中直接实现了上面给出的计算公式，表达式的右边由 k 个因子组成，因而它用一个迭代 k 次的循环来进行计算：

```
public static int c( int n, int k )
{
  int c =1;
  for ( int j =1; j <=k; j++ )
    c = c ×(n - j + 1) / j;
return c;
}
```

4.5 欧几里德算法

欧几里德算法计算两个正整数的最大公约数，出现在欧几里德原本（Euclid's Elements）第 7 卷命题 2 中，正如欧几里德最初所表述的一样，它反复从较大的数 m 中减去较小的数 n，直到差 d 比 n 小为止，然后，让 d 扮演 n 的角色，而 n 扮演 m 的角色，即反复从较大的数 n 中减去较小的数 d。重复上述过程，直至差与减数相等为止，这是的减数就是原先两数间的最大公约数。

```
494
-130
364
-130
234
-130   130
104   -104   104
        26   -26
              78
             -26
              52
             -26
              26
```

右边的例子就利用欧几里德算法来找到 494 和 130 间的最大公约数 26，因为 494=26 × 19，130=26 ×5，所以，获得的结果是正确的。

例 4.13 欧几里德算法的递归实现

算法中的每一步简单地从较大的数中减去较小的数，这可通过递归调用 gcd(m , $n-m$) 或 gcd ($m-n$, n)来完成。

```
Public static ing gcd ( int m , int n )
{  if ( m == n ) return n ;                          //递归基
   else if (m <n ) return  gcd ( m , n - m );        //递归体部分
   else return gdc ( m - n , n );
}
```

例如，调用 gcd (494 , 130)递归调用 gcd (364 , 130)，它又递归调用 gcd (234 , 130)，

标准分享网 www.bzfxw.com 免费下载

而它又调用 gcd (104 , 130)，再次调用 gcd (104 , 26)，它又调用 gcd (78 , 26)，随后调用 gcd (52 , 26)，再递归调用 gcd (26 , 26)，该方法将返回 26，该值然后不断地沿返回链返回到最初的调用 gcd (494 , 130)，它将该值返回给它的调用者。

4.6 正确性的归纳证明

递归函数的正确性常用数学归纳法的原理来进行证明（见附录 B），该原理称一个命题的无限序列可以通过证实以下两部分来证明原命题为真命题。这两部分是：(1) 命题序列中的初始部分为真，(2)命题序列中其他部分为真，可以在假定它的前面的命题部分均为真的条件下进行推导，如能推导为真，则原命题为真。第一部分也称为初始归纳步，而第二部分称为归纳步，而对先前命题部分为真的假设称为归纳假设。

定理 4.2 递归阶乘函数是正确的

证明：为证明阶乘函数的递归实现是正确的（例 4.2 ）先证明初始归纳步是正确的。调用 $f(0)$ 将返回正确的值 1，这是由于第一行 if (n <2) return 1 ; 的缘故。

接下来，我们假定对某个 n>0 小于 n 的所有整数而言，递归阶乘函数均将返回正确的值，此时，第二行 return $n \times f(n-1)$; 将返回正确的 n !值，这是由于从归纳假设可知，$f(n-1)$ 将返回正确的值 $(n-1)!$，而 $n! = n \times (n-1)!$。这正是递归阶乘函数所返回的结果。

注意，这里所使用的数学归纳法为强数学归纳法（也称第二数学归纳法），在该原理中，归纳假设允许我们假定先前的所有语句均为真。而在弱（或第一数学归纳法）归纳法中，仅允许假定前一个语句为真。但这两个原理是相等的，也就是说，两个均是有效的证明方法，这样，应用强归纳原理通常更好。

定理 4.3 欧几里德算法是正确的

证明：我们可以使用强数学归纳原理来证明欧几里德算法是正确的（例 4.13）。假如 m 和 n 相等，则该数就是这两个数中的最大公约数，因此，方法返回正确的值，这是由于以下程序代码所获得的：

$$if(m == n)\ \text{return}\ n;$$

假如 m 和 n 不相等，这样，函数要么返回 gcd (m , $n-m$)，要么返回 gcd ($m-n$, n)。要确定这个返回值也是正确的，我们仅需要确定(m , n) ，(m , $n-m$)，($m-n$, n)这 3 对数总是具有相同的最大公约数。而这是数论中的一个定理，在附录 B 中给出了证明。

4.7 递归算法的复杂性分析

递归算法的复杂性分析依赖于相应的循环关系（recurrence relation）的可解性，求解循

环关系的一般技术是令 $T(n)$表示算法在问题规模为 n 时所执行的步数，算法的递归体部分转变成 $T(n)$上的循环关系，该循环关系的解则为算法的复杂性函数。

定理 4.4 递归阶乘函数的运行时间为 $O(n)$

证明：令 $T(n)$为例 4.2 中从最初调用 $f(n)$时所产生的递归调用的次数，则 $T(0)=T(1)=0$，这是因为当 $n<2$ 时，将不进行递归调用。假如 $n>1$，则程序行 return n * $f(n-1)$将被执行，将产生递归调用 $f(n-1)$，将它转变为以下循环关系：

$$T(n)=1+T(n-1)$$

上述循环关系的解为：

$$T(n)=n-1, for \quad n>0$$

这一结论由两阶段得到：首先找到一个解，然后使用归纳法来证明该解是正确的。

为找到循环关系的解，一种最简单的技术就是制作一张值表，如表 4.3，并从中发现解模式。

表 4.3 问题规模与递归调用次数关系表

n	$T(n)$
0	0
1	0
2	1
3	2
4	3
5	4
6	5

上述循环关系说明 $T(n)$的每一个值均比前一个值多一，因此解 $f(n)=n-1$ 就是相当明显的了。

现在来证明 $T(n)=n-1$ 对所有 $n>0$ 时的正确性。令 $f(n)=n-1$，并应用弱数学归纳原理来对此进行证明。选择归纳基为 $n=1$，此时，$T(n)=T(1)=0$ 以及 $f(n)=f(1)=(1)-1=0$。在归纳步，假定 $T(n)=f(n)$对某个 $n>1$ 时成立，由此推演出假设所应有的结论 $T(n+1)=f(n+1)$。

$$T(n+1)=1+T(n)=1+f(n)=1+(n-1)=n$$

$$f(n+1)=(n+1)-1=n$$

这就完成了定理的证明。

至此，我们确定了阶乘函数的递归实现的复杂性函数为 $T(n)=n-1$，因而可以得出结论，该实现的运行时间为 $O(n)$。这意味着执行时间与参数 n 的大小成正比，如果计算 8!的时间为 3 毫秒，则计算 16 !的时间为 6 毫秒。

4.8 动态规划

在多数情况下，由于频繁的函数调用使得递归算法非常低效，因此，当迭代实现不太复杂时，用迭代实现的方式来实现更好。另一种实现循环关系的方式是用一个数组来保存先前的计算结果，而不像递归时那样进行重复计算。这种方法称为动态规划。

标准分享网 www.bzfxw.com 免费下载

例 4.14 斐波那契函数的动态递归实现

```
public class Ex0414
{ public static int fib(int n)
  { if (n<2) return n;
    int[] f = new int[n];
    f[0] = 0;
    f[1] = 1;
    for (int i=2; i<n; i++)    // store the Fibonacci numbers
      f[i] = f[i-1] + f[i-2];
    return f[n-1] + f[n-2];
  }

  public static void main(String[] args)
  { for (int n=0; n<16; n++)
      System.out.println("fib("+n+") = "+fib(n));
  }
}
```

在上述实现中使用了一个 n 个整数的动态数组 f[] 来保存前 n 个斐波那契数。

4.9 汉内塔

我们已经看到了一些使用递归的重要例子，用递归来进行定义更为自然，理解起来更加容易。对某些问题而言，递归是惟一合理的表达方法。

汉内塔游戏是一个经典的需要用递归来进行求解的例子，该游戏由一块水平板和钉在水平板上的三根直柱组成，并分别用 A，B，C 标注，以及 n 个在中心有小孔的园盘，园盘的半径成算术级数进行增加（如 5 厘米，6 厘米，7 厘米，8 厘米，...），并套在 A 柱上，如图 4.3 所示。游戏的规则是套在同一根柱上的盘片不允许半径大的压半径小的。游戏的目标是把套在 A 柱上的盘片移动到 C 柱上，在挪动过程中，不能违反游戏规则。

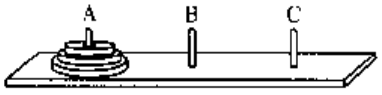


图 4.3 汉内塔游戏示意图

汉内塔游戏的一般解法是一个自然的递归过程。

第一步：将 A 柱上面的 n-1 个盘从 A 柱移到 B 柱。

第二步：将剩余的那个盘从 A 柱移到 C 柱。

第三步：将 B 柱上的 n -1 个盘从 B 柱移到 C 柱。

上面的第一步和第三步均是递归的将完全解（n 个盘）应用到 n -1 个盘的情形。该递归求解的基为 n = 0 时的情况，在这种情况下，无需进行其他工作。

在 n = 1 时的求解过程是：

将盘从 A 柱移到 C 柱。

$n=2$ 时的求解过程是：

1. 将顶上的盘从 A 柱移到 B 柱。
2. 将剩余的那个盘从 A 柱移到 C 柱。
3. 将 B 柱上的盘移到 C 柱。

$n=3$ 时的求解过程是：

1. 将 A 柱顶上的盘移到 C 柱。
2. 将 A 柱上的盘移到柱 B。
3. 将 C 柱顶上的盘移到 B 柱。
4. 将 A 柱顶上的盘移到 C 柱。
5. 将 B 柱顶上的盘移到 A 柱。
6. 将 B 柱上的盘移到 C 柱。
7. 将 A 柱顶上的盘移到 C 柱。

在上述过程中，前 1～3 小步可以看成汉内塔游戏的一般解法 3 个步骤中的第一步，而第 4 小步为一般解法中的第二步，第 5～7 小步为一般解法中的第三步。

因为一般解法中需要对替换不同的柱标志，因此使用变量更好。将三步汉内塔算法表示为 hanoi(n, x, y, z)，它变成：

第一步：将柱 x 上较小的 n－1 个盘移到柱 z。

第二步：将柱 x 上的剩余的盘移到柱 y。

第三步：将柱 z 上的 n－1 个盘移到柱 y。

该通用算法将在例 4.15 中实现。

例 4.15 汉内塔

```
public class Ex0415
{ public static void main(String[] args)
  { runHanoi(4,'A','B','C');
  }

  public static void runHanoi(int n, char x, char y, char z)
  { if (n==1)  // basis

      System.out.println("Move top disk from peg " + x
                    + " to peg " + z);
    else      // recursion
    { runHanoi(n-1,x,z,y);
      runHanoi(1,x,y,z);
      runHanoi(n-1,y,x,z);
    }
  }
```

标准分享网 www.bzfxw.com 免费下载

```
}
```

求解有4个盘的汉内塔游戏可以通过调用 runHanoi (4, 'A', 'B', 'C'); 来获得：其输出为：

```
Move top disk from peg A to peg B
Move top disk from peg A to peg C
Move top disk from peg B to peg C
Move top disk from peg A to peg B
Move top disk from peg C to peg A
Move top disk from peg C to peg B
Move top disk from peg A to peg B
Move top disk from peg A to peg C
Move top disk from peg B to peg C
Move top disk from peg B to peg A
Move top disk from peg C to peg A
Move top disk from peg B to peg C
Move top disk from peg A to peg B
Move top disk from peg A to peg C
Move top disk from peg B to peg C
```

4.10 互递归

当一个函数调用它自己时，称为直接递归。另一种递归形式是一个函数调用其他函数，而这些函数最终将调用最开始的那个函数，这称为间接递归。间接递归的最常见形式是两个函数间的相互调用，称这种调用关系为互递归 (mutual recursion)。这三种递归调用关系如图4.4所示。

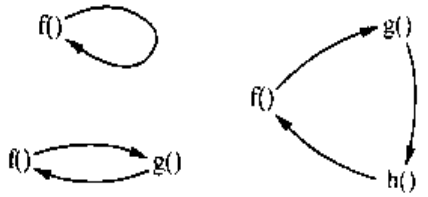


图 4.4 三种递归调用关系

例 4.16 用互递归计算正弦和余弦函数

三角形函数中的正弦和余弦函数可以有好几种定义形式，计算这些函数值的算法也有好几个，最为简单的算法就是通过互递归来进行计算（尽管不是最有效的方式）。它基于一些等式：

$$\sin 2\theta = 2\sin\theta\cos\theta$$

$$\cos 2\theta = 1 - 2(\sin\theta)^2$$

以及以下两个泰勒多项式展开：

$$\sin x \approx x - x^3/6$$

$$\cos x \approx 1 - x^2/2$$

上两个展开公式在 x 较小的情况下是非常接近实际值的。

```
public class Ex0416
{ public static void main(String[] args)
  { for (double x=0. ; x<1.0; x += 0.1)
      System.out.println(s(x) + "\t" + Math.sin(x));
    for (double x=0. ; x<1.0; x += 0.1)
      System.out.println(c(x) + "\t" + Math.cos(x));
  }
  public static double s(double x)
  { if (-0.005 < x && x < 0.005) return x - x*x*x/6;  // basis
    return 2*s(x/2)*c(x/2);                          // recursion
  }
  public static double c(double x)
  { if (-0.005 < x && x < 0.005) return 1.0 - x*x/2;  // basis
    return 1 - 2*s(x/2)*s(x/2);                      // recursion
  }
}
```

上述程序的输出是：

```
0.0     0.0
0.09983341664635367     0.09983341664682815
0.1986693307941265      0.19866933079506122
0.29552020665442036     0.2955202066613396
0.3894183423068936      0.3894183423086505
0.4794255385990945      0.479425538604203
0.5646424733830799      0.5646424733950354
0.6442176872361944      0.644217687237691
0.7173560908968648      0.7173560908995227
0.7833269096232099      0.7833269096274833
0.8414709848016061      0.8414709848078964
1.0     1.0
0.9950041652780733      0.9950041652780258
0.9800665778414311      0.9800665778412416
0.9553364891277464      0.955336489125606
0.9210609940036278      0.9210609940028851
0.8775825618931635      0.8775825618903728
0.8253356149178575      0.8253356149096783
0.7648421872857489      0.7648421872844885
0.6967067093499022      0.6967067093471655
0.6216099682760496      0.6216099682706645
0.5403023058779364      0.5403023058681398
```

上述程序中之所以得出非常相近的结果，这是因为它使得 x 多次递归被 2 除，一直到达递归基标准（-0.005 < x && x < 0.005）时才终止递归，当 x 如此之小时，泰勒多项式展开获得的结果非常准确，精确到小数点 15 位。

标准分享网 www.bzfxw.com 免费下载

复 习 题

4.1 一个递归函数必须要有两部分：递归基和递归体，请对它们进行解释。为什么说它们对递归而言，是至关重要的？

4.2 调用递归阶乘函数 f(10)将产生多少次递归调用（见例 4.2）？

4.3 调用递归斐波那契函数 fib(6)将产生多少次递归调用（见例 4.2）？

4.4 采用递归求解而不是迭代求解的优点和不足是什么？

4.5 直接递归和间接递归间的不同点是什么？

习 题

4.1 编写并测试一个返回前 n 个整数的平方和的递归函数。

4.2 编写并测试一个返回 b 的 1 到 n 次幂的和的递归函数。

4.3 编写并测试一个返回数组前 n 个元素的和的递归函数。

4.4 编写并测试一个返回数组前 n 个元素中最大元素的递归函数。

4.5 编写并测试一个返回数组前 n 个元素中最大元素的递归函数，该函数最多使用 $\lg n$ 次递归调用。

4.6 编写并测试一个返回 x 的 n 次幂的递归函数。

4.7 编写并测试一个返回 x 的 n 次幂的递归函数，该函数最多使用 $2\lg n$ 次递归调用。

4.8 编写并测试一个返回整数 n 的以 2 为底的对数的整数部分（也即 n 可以被 2 除的次数）。

4.9 编写并测试一个递归布尔函数，该函数确定一个字符串是否为一个回文字串（一个回文字串是一个将其字符逆转后不变的文字串）。

4.10 编写并测试一个将十进制整数转换为二进制表示的递归函数，该函数返回该二进制数的字符串表示。

4.11 编写并测试一个将十进制整数转换为十六进制表示的递归函数，该函数返回该十六进制数的字符串表示。

4.12 编写并测试一个打印字符串中前 n 个字符的所有置换的递归函数。例如，调用 `print( "ABC" )`将打印：

```
ABC
ACB
BAC
BCA
CBA
CAB
```

4.13 用迭代方式实现斐波那契函数（不使用数值）。

4.14 实现阿克曼函数（Ackermann function）。

$$a(0,n)=1$$
$$a(1,0)=2$$
$$a(m,0)=m+2, \text{ if } m>1$$
$$a(m,n)=a(a(m-1,n),n-1), \quad \text{if } m>0 \text{ and } n>0$$

4.15 证明帕斯卡循环关系的正确性（见4.4节）。

4.16 跟踪递归实现的欧几里德算法（例4.6）对调用gcd(385,231)时所生成的递归过程。

4.17 以迭代方式实现欧几里德算法。

4.18 利用整数模算子%而不用重复进行减法操作的方式来实现递归欧几里德算法。

4.19 利用整数模算子%而不用重复进行减法操作，并用迭代的方式来实现欧几里德算法。

4.20 使用数学归纳法来证明斐波那契函数的递归实现（例4.6）是正确的。

4.21 使用数学归纳法来证明习题4.4中的递归函数是正确的。

4.22 使用数学归纳法来证明习题4.5中的递归函数是正确的。

4.23 使用数学归纳法来证明习题4.8中的递归函数是正确的。

4.24 使用数学归纳法来证明习题4.12中的递归函数是正确的。

4.25 一个函数的可计算域是能使函数产生正确结果的那个输入集，用实验的方式确定例4.2中所实现的阶乘函数的可计算域。

4.26 用实验的方式确定习题4.2中所实现的sum(b,n)函数的可计算域，假定$b=2$。

4.27 用实验的方式确定习题4.3中所实现的斐波那契函数的可计算域。

4.28 用实验的方式确定例4.11中所实现的计算递归二项式系数的函数的可计算域。

4.29 汉内塔程序对有3个盘的情况时将一共进行7次盘的移动，请问对以下情况将产生多少次盘的移动？

a. 5个盘。

b. 6个盘。

c. n个盘。

4.30 请证明前一习题中所导出的公式。

4.31 用实验的方式确定习题4.14中所实现的阿克曼函数的可计算域。

4.32 显示习题4.15中调用hanoi(4,'A','B','C')时所产生的递归调用树。

4.33 修改例4.16中的程序，使其递归基更窄，递归直至|x|<0.000 05，从而使得结果更为精确。

4.34 修改例4.16中的程序，用更为精确的近似计算公式：

$$\sin x \approx x - x^3/6 + x^5/120 = x(1 - x^2/6(1 - x^2/20))$$

$$\cos x \approx 1 - x^2/2 + x^4/24 = 1 - x^2/2.(1 - x^2/12)$$

4.35 使用互递归实现双曲正弦和余弦函数，使用的公式为：

标准分享网 www.bzfxw.com 免费下载

$$\sinh 2\theta = 2\sinh\theta\cosh\theta$$

$$\cosh 2\theta = 1 + 2(\sinh\theta)^2$$

$$\sinh x \approx x + x^3/6$$

$$\cosh x \approx 1 + x^2/2$$

将你的结果与以下用指数函数定义的相应的 sinhx 和 coshx 值进行比较。

$$\sinh x = \frac{(e^x - e^{-x})}{2}$$

$$\cosh x = \frac{(e^x + e^{-x})}{2}$$

4.36 利用以下递归公式实现正弦函数：

$$\tan 2x = \frac{2\tan x}{1-(\tan x)^2}$$

$$\tan x \approx x + \frac{1}{3}x^3$$

4.37 实现一个求解多项式 $a_0 + a_1x + a_2x^2 + \ldots + a_nx^n$ 的递归函数，其中的 $n+1$ 个系数 a_i 将通过一个数组以及指明该数组的度的方式传递给该函数。

复习题答案

4.1 递归函数的递归基在递归函数的定义中是它的起点，当被递归调用时又是它的最后终止步，该条件将使递归终止。递归函数的递归体部分出现在赋值语句中，赋值语句的右边同样出现该函数，因此导致自己调用自己，这样就导致重复出现。例如，在阶乘函数中，递归基是 $n! = 1$ (当 n 为 0)，而递归体部分为 $n! = n(n-1)!$ (当 $n > 0$)。

4.2 调用 factorial(10)将产生 10 次递归调用。

4.3 调用斐波那契函数 $f(6)$ 将产生 $14 + 8 = 22$ 次递归调用，这是由于 $f(6)$ 将调用 $f(5)$ 和 $f(4)$，而 $f(5)$ 和 $f(4)$ 将分别产生 14 和 8 次递归调用。

4.4 一个递归解通常比一个迭代解更易于理解，但递归解的运行速度通常比迭代解慢。

4.5 直接递归是一个函数直接调用其自身，而间接递归是一组函数，它们之间相互调用。

习 题 解 答

4.1 返回前 n 个数的平方和的递归函数为：

```
public static int sum(int n)
{ if (n==0) return 0;     // basis
  return sum(n-1) + n*n;  // recursion
```

```
}
```

4.2 返回b的1到n次幂的和的递归函数。

```
public static double sum(double b, int n)
{ if (n==0) return 1;       // basis
  return 1 + b*sum(b,n-1);  // recursion
}
```

4.3 返回数组前n个元素的和的递归函数。

```
public static double sum(double[] a, int n)
{ if (n==0) return 0.0;       // basis
  return sum(a,n-1) + a[n-1];  // recursion
}
```

4.4 返回数组前n个元素中最大的那个元素的递归函数。

```
public static double max(double[] a, int n)
{ if (n==1) return a[0];       // basis
  double m = max(a,n-1);       // recursion
  if ( a[n-1] > m )  return a[n-1];
  else return m;
}
```

4.5 返回数组前n个元素中最大的那个元素的递归函数，该函数使用的递归调用不多于lg n次。

```
public static double max(double[] a, int n)
{if (n==1) return a[0];          // basis
 int n1 = n/2;
 int n2 = n - n1;
 double m1 = max(a,n1);          // recursion on a [0:n1-1]
 double m2 = max(a+n1,n2);       // recursion on a [n1:n-1]
 return (m1 > m2 ? m1 : m2);     // = max{m1,m2}
}
```

4.6 一个返回x的n次幂的递归函数。

```
public static double pow(double x, int n)
{ if (n==0) return 1.0;    // basis
  return x*pow(x,n-1);     // recursion
}
```

4.7 一个返回x的n次幂的递归函数，该函数最多使用$2lg^n$次递归调用。

```
public static double pow(double x, int n)
{ if (n==0) return 1.0;    // basis
  double p =pow(x,n/2) ;
```

标准分享网 www.bzfxw.com 免费下载

```
  if ( n%2 == 0 ) return p*p ; // recursion (n even)
  else return  x*p*p;          // recursion (n odd)
}
```

4.8 一个返回整数 n 的以2为底的对数的整数部分（也即 n 可以被2除的次数）。

```
public static int lg(int n)
{ if (n==1) return 0 ;    // basis
  return  1 + lg(n/2) ;         // recursion
}
```

4.9 确定一个字符串是否为回文字串递归布尔函数。

```
public static boolean isPalindrome(String s)
{ // returns true iff the reverse of s is the same as s
  int len = s.length();
  if (len<2) return true;
  if (s.charAt(0) != s.charAt(len-1)) return false;
  if (len==2) return true;
  return isPalindrome(s.substring(1,len-1));
}
```

4.10 一个将十进制整数转换为二进制表示的递归函数。

```
public static String binary(int n)
{  String s = "" + n%2;
   if (n < 2) return s ;              // basis
   return  binary(n/2) + s;  // recursion
}
```

4.11 一个将十进制整数转换为十六进制表示的递归函数。

```
public static String hexadecimal(int n)
{  String s = hex(n%16);
   if (n < 15) return s ;              // basis
   return  hexadecimal(n/16) + s;  // recursion
}
String hex(int n)
{ if (n = = 0) return "0" ;
  if (n = = 1) return "1" ;
  if (n = = 2) return "2" ;
  if (n = = 3) return "3" ;
  if (n = = 4) return "4" ;
  if (n = = 5) return "5" ;
  if (n = = 6) return "6" ;
  if (n = = 7) return "7" ;
  if (n = = 8) return "8" ;
```

```
    if (n = = 9) return "9" ;
    if (n = = 10) return "A" ;
    if (n = = 11) return "B" ;
    if (n = = 12) return "C" ;
    if (n = = 13) return "D" ;
    if (n = = 14) return "E" ;
      else                return "F";
    }
```

4.12 打印置换的递归函数。

```
Public static void pint(String str)
{ print("",str);
}

public static void print(String left,String right)
{ int n=right.length()
  if (n==0) return;
  if (n==1)
  { System.out.println(left+right);
    return;
  }
  StringBuffer s = new StringBuffer(right);
  for (int i=0;i<n;i++)
  { char temp = s.charAt(i);
    s.setCharAt(i,s.charAt(0));
    s.setCharAt(0,temp);
    print(left+temp,s.substring)(1,n);
  }
 }
}
```

4.13 斐波那契函数的迭代实现。

```
public static int fib(int n)
{  if (n< 2) return n ;
   int f0 = 0, f1 = 1, f=f0+f1;
   for ( int i=2; i<=n; i++)
   { f0 = f;
     f1 = f;
     f = f0 + f1 ;
   }
   return f;
}
```

4.14 阿克曼函数。

标准分享网 www.bzfxw.com 免费下载

```
public static int ackermann(int m, int n)
{  if (m == 0) return 1 ;        // 递归基础
   if (n==0)
     if (m==1) return 2 ;        // 递归基础
     else return m + 2 ;         // 递归基础
   int ackM = ackermann(m-1,n) ;
   return ackermann(ackM, n-1); // 循环体
}
```

4.15 证明帕斯卡循环关系的正确性。

先考虑 $(x+1)^8$ 的展开式中的关系 $c(8,3)=56=35+21=c(7,3)+c(7,2)$。

$$
\begin{aligned}
(x+1)^8 &= (x+1)(x+1)^7 \\
&= (x+1)(x^7+7x^6+21x^5+35x^4+35x^3+21x^2+7x+1) \\
&= x^8+7x^7+21x^6+35x^5+35x^4+21x^3+7x^2+x \\
&\quad +x^7+7x^6+21x^5+35x^4+35x^3+21x^2+7x+1 \\
&= x^8+8x^7+28x^6+56x^5+70x^4+56x^3+28x^2+7x+1
\end{aligned}
$$

系数 $c(8,3)$ 是用于项 x^5 的系数，它为 $35x^5+21x^5=56x^5$。而和 $35x^5+21x^5$ 从 $x(35x^4)$ 和 $(1)(21x^5)$ 而来。因此这些系数为 $35=c(7,3)$，$21=c(7,2)$。

一般性的证明基于同样的讨论，$c(n,k)$ 是展开式 $(x+1)^n$ 中项 x^k 的系数，因为：$(x+1)^n=(x+1)(x+1)^{n-1}$，所以该项来源于以下两项之和。

$(x)(c(n-1,k-1)x^{k-1})+(1)(c(n-1,k)x^k)=(c(n-1,k-1)+c(n-1,k))x^k$

因此，$c(n,k)=c(n-1,k-1)+c(n-1,k)$。

4.16 gcd(385,231)的调用关系如图 4.5 所示。

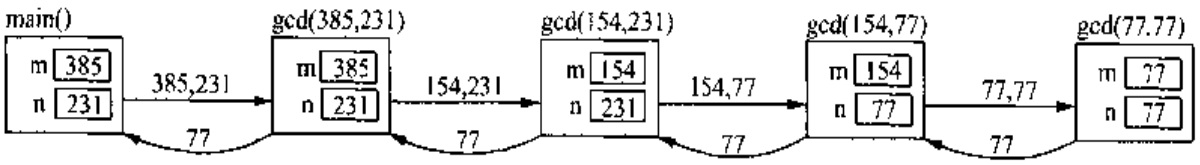


图 4.5 调用关系图

4.17 欧几里德算法的迭代实现。

```
public static int gcd(int m, int n)
{ while (m != n)              //INVARIANT: gcd(m,n)
    if (m < n) n -= m;
    else m -= n;
  return n ;
}
```

4.18 利用整数模算子的递归欧几里德算法实现。

```
public static int gcd(int m, int n)
{ if (m==0) return n ;          // basis
```

```
  if (n==0) return m ;          // basis
  if (m < n) return gcd(m,n%m) ; // recursion
  else return gcd(n,m%n) ; // recursion
}
```

4.19 利用整数模算子的迭代欧几里德算法实现。

```
public static int gcd(int m, int n)
{ while( n > 0 )  // 不变式: gcd (m , n)
 { int r = m%n;
  m = n;
  n = r;
 }
 return m ;
 }
```

4.20 使用数学归纳法来证明斐波那契函数的递归实现是正确的。首先证明它的递归基，调用 fib(0)和 fib(1)将返回正确的值 0 和 1，因为以下代码行：

if (n < 2) return n;

所保证的。下面，假定对某个大于 1 的数 n 而言，函数返回正确的值，则第二行：

return fib (n –1) + fib (n– 2) ;

将返回正确的 fib(n)值，这是由于从归纳假设可知：调用 fib (n –1)和 fib (n– 2) 将分别返回正确的值 F_{n-1} 和 F_{n-2}，而 $F_n = F_{n-1} + F_{n-2}$ 正是定义所确定的值。

注意：递归基中应当包括 n 为 0 和为 1 的情况，这是因为 $F_n = F_{n-1} + F_{n-2}$ 公式只对 $n > 1$ 的情况才能应用。

4.21 如果 n= 1，递归基将执行，返回 a [0]。这是数组中的最大元素，因为这是数组中的惟一的元素，所以也为最大。

假如 $n > 1$ 时函数可以计算出前 n –1 个元素中的最大元素（归纳假设），此时如果条件（a [n –1] > m ）为真，则将返回元素 a [n –1]，这将是 n 个元素中最大的元素，因为它大于 m，而 m 又是前 n– 1 个元素中的最大值。当条件（a [n –1] > m ）为假时，返回 m，它是所有 n 个元素中的最大值。这是由于它不比 a [n –1]小，且是数组中前 n –1 个元素中的最大元素。

4.22 当 n =1 时，递归基将被执行，返回 a[0]，这是数组中的最大元素，因为此时它是数组中的惟一元素。

假如 $n > 1$ 时，函数正确计算出数组的第一和第二部分的最大值 $m1$ 和 $m2$（归纳假设），则这两个中的最大的一个为整个数组的最大值，将其返回，因而获得正确值。

4.23 当 n = 1 时，递归基被执行，返回 0，这就是 n 可被 2 除的次数。

假如对任何小于 n（n >1）数而言，函数均能得到正确结果，则 n /2 可被 2 除的次数可以正确得到（由归纳假设可知为 lg (n /2)），它比 n 可被 2 除的次数小 1，因此，对 n 而言所返回的值 1 + lg (n /2)是正确的。

标准分享网 www.bzfxw.com 免费下载

4.24 首先，我们来证明一个猜想：调用 print(left, right)将打印出 $n!$个不同的字符串，所有这些字符串的前缀均为 left，n = right.legth()。

当 n =1 时，print(left, right)将打印出 left + right 并返回，这就是 1!个不同的字符串。假如 right.legth() = n–1 时，调用 print(left, right)将打印出(n – 1)!个不同的并以 left 为前缀的字符串，则当 right.legth() = n 时，for 循环将产生 n 次形为 print(left + temp, ss) 的调用，其中的 temp 为一个不同的字符，而 ss = s.substring(1, n)。因为 s.substring(1, n) 的长度为 n–1，因此，每次调用都将产生具有相同前缀 left+ temp 的（n–1）!个不同的字符串。因此，循环将产生 n (n–1)!个不同的具有相同的 left 前缀的字符串。这就用归纳法证明了上述猜想。

从猜想中可知，调用 print(str)将产生 $n!$个不同的字符置换，n 为 str 的长度。因为这正好是该字符串所能有的置换的数目，因此，该方法产生的结果是正确的。

4.25 对例子 4.2 中的阶乘函数实现而言，整数的溢出发生在当 n = 13 时的 long 返回类型值上（对作者的机器而言），因此，该函数的可计算域为 $0 \leqslant n \leqslant 12$。

4.26 对习题 4.2 中所实现的 sum (b , n)函数而言，浮点数溢出发生在 n = 1 023 时的 double 返回类型值上（对作者的机器而言），因此，该函数的可计算域为 $0 \leqslant n \leqslant 1\,022$。

4.27 对习题 4.3 中所实现的斐波那契函数而言，在 n=36 以后，递归产生的额外负担使得程序的效率明显降低（在作者的机器上），因此，该函数的可计算域大约为 $0 \leqslant n \leqslant 40$。

4.28 对例 4.11 中所实现的计算递归二项式系数的函数而言，在 n = 25 以后，递归产生的额外负担使得程序的效率明显降低（在作者的机器上），因此，该函数的可计算域大约为 $0 \leqslant n \leqslant 30$。

4.29 汉内塔程序对有 3 个盘的情况时将一共进行 7 次盘的移动，请问对以下情况将产生多少次碟的移动?

a. 在 5 个盘时的移动次数为 31 次。

b. 在 6 个盘时的移动次数为 63 次。

c. n 张碟时的移动次数为 $2^n - 1$。

4.30 为了证明汉内塔程序在有 n 个盘时将进行 $2^n - 1$ 次盘的移动，我们使用数学归纳法。归纳基已在例子 4.15 中建立，为了移动 n + 1 个盘，将首先进行 $2^n - 1$ 次碟的移动把上面的 n 个盘移到柱 B（由归纳假设），需要一次将柱 A 上的最底的那个盘移到柱 C，然后用 2^n–1 次将柱 B 上的 n 个盘移到柱 C，因此，整个移到次数为$(2^n - 1) + 1 + (2^n - 1) = 2^{n+1} - 1$。

4.31 对习题 4.14 中所实现的阿克曼函数而言，在（m =17，n = 2）、（m =5，n = 3）、（m =4，n = 4）以及（m =3，n = 5）等条件下均产生异常，因此，该函数的可计算域限制在：

当 n =2 时，$0 \leqslant m \leqslant 16$;

当 n =3 时，$0 \leqslant m \leqslant 4$;

当 n =4 时，$0 \leqslant m \leqslant 3$;

当 n =5 时，$0 \leqslant m \leqslant 2$。

4.32 显示习题4.15中调用hanoi(4, 'A', 'B', 'C')时所产生的递归调用树，如图4.6所示。

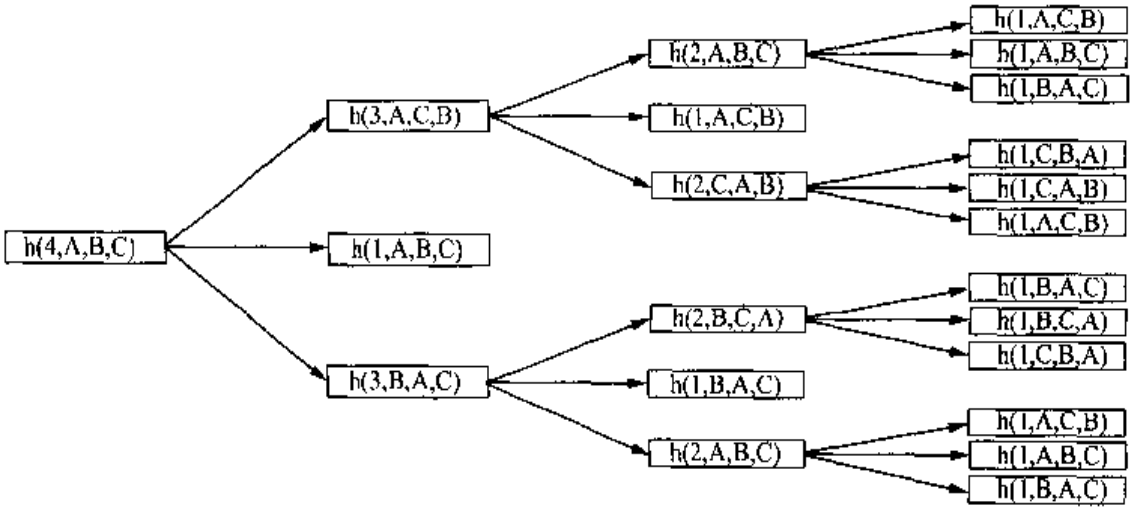


图4.6 习题4.32中的递归调用树

4.33 sin和cos函数的更为精确的递归实现。

```
public class Pr0433
{ public static void main(String[] args)
  { for (double x=0. ; x<1.0; x += 0.1)
      System.out.println(s(x) + "\t" + Math.sin(x));
    for (double x=0. ; x<1.0; x += 0.1)
      System.out.println(c(x) + "\t" + Math.cos(x));
  }
  public static double s(double x)
  { if (Math.abs(x) < 0.00005) return x - x*x*x/6 ;
    return 2*s(x/2)*c(x/2);
  }
  public static double c(double x)
  { if (Math.abs(x) < 0.00005)
       return 1.0 - x*x/2;
    return 1 - 2*s(x/2)*s(x/2);
  }
}
```

4.34 利用更为精确的近似计算公式来计算sin和cos函数。

```
public class Pr0434
{ public static void main(String[] args)
  { for (double x=0. ; x<1.0; x += 0.1)
      System.out.println(s(x) + "\t" + Math.sin(x));
    for (double x=0. ; x<1.0; x += 0.1)
      System.out.println(c(x) + "\t" + Math.cos(x));
```

标准分享网 www.bzfxw.com 免费下载

```
  }
  public static double s(double x)
  { if (Math.abs(x) < 0.005)
     return x*(1 - x*x/6*(1-x*x/20)) ;          // 递归基础
    return 2*s(x/2)*c(x/2);          // 递归体部分
  }
  public static double c(double x)
  { if (Math.abs(x) < 0.005)
       return 1.0 - x*x/2*(1-x*x/12);          // 递归基础
    return 1 - 2*s(x/2)*s(x/2);       // 递归体
  }
}
```

4.35 双曲线正弦和余弦函数的互递归实现。

```
public class Pr0435
{ public static void main(String[] args)
  { for (double x=0. ; x<1.0; x += 0.1)
     System.out.println(s(x) + "\t" + sinh(x));
    for (double x=0. ; x<1.0; x += 0.1)
     System.out.println(c(x) + "\t" + cosh(x));
  }
  public static double s(double x)
  { if (Math.abs(x) < 0.005)
     return x*(1 + x*x/6) ;
    return 2*s(x/2)*c(x/2);
  }
  public static double c(double x)
  { if (Math.abs(x) < 0.005)
       return 1.0 + x*x/2;
    return 1 + 2*s(x/2)*s(x/2);
  }
  //nonRecursive methods
  public static double sinh(double x)
  { return (Math.exp(x) - Math.exp(-x))/2;
  }
  public static double cosh(double x)
  { return (Math.exp(x) + Math.exp(-x))/2;
  }
}
```

4.36 正弦函数的递归实现。

```
public class Pr0436
  { double pi4 = Math.PI / 4.0 ;
    System.out.println("t(pi/4) = " + t(pi4)
```

```
        + "  Math.tan(pi/4) = " + Math.tan(pi4) ) ;

    for (double x=0. ; x<1.0; x += 0.1)
      System.out.println(t(x) + "\t" + Math.tan(x));
  }
  public static double t(double x)
  { if (Math.abs(x) < 0.0005)
      return x*(1 + x*x/3) ;          // basis
    double half = t(x/2) ;
    return 2*half/(1.0-half * half) ;  // recursion
  }
}
```

4.37 多项式函数的递归求值。

```
private static double p(double[] a, double x, int k)
{ // returns a[k] + a[k+1]*x + a[k+2]*x*x + ...
  if (k == a.length) return 0;  // 递归基础
  return a[k] + x*p(a,x,k+1);   // 递归体
```

标准分享网 www.bzfxw.com 免费下载

第5章 汇　集

汇集(collection)是一种对象包容器，在 Java 1.2 中，用汇集框架的接口和类对这个概念进行了规范。

5.1 Java 汇集框架

在 java.util 包中定义了用于汇集的以下 8 个接口框架，如表 5.1 所示。

表 5.1 汇集接口框架

接口	描述
Collection	元素的汇集
List	元素的序列
Set	没有重复元素的汇集
SortedSet	没有重复元素且排序的汇集
Map	元素为二元偶对的汇集，偶对形式为（key,value）其中的 key 必需惟一，在汇集中不能重复
SortedMap	在 Map 的基础上，加上排序的要求
Iterator	一个可遍历总个汇集的对象
ListIterator	可遍历一个序列 List 的对象

图 5.1 表明了这些接口和实现这些的类之间的相互关系。

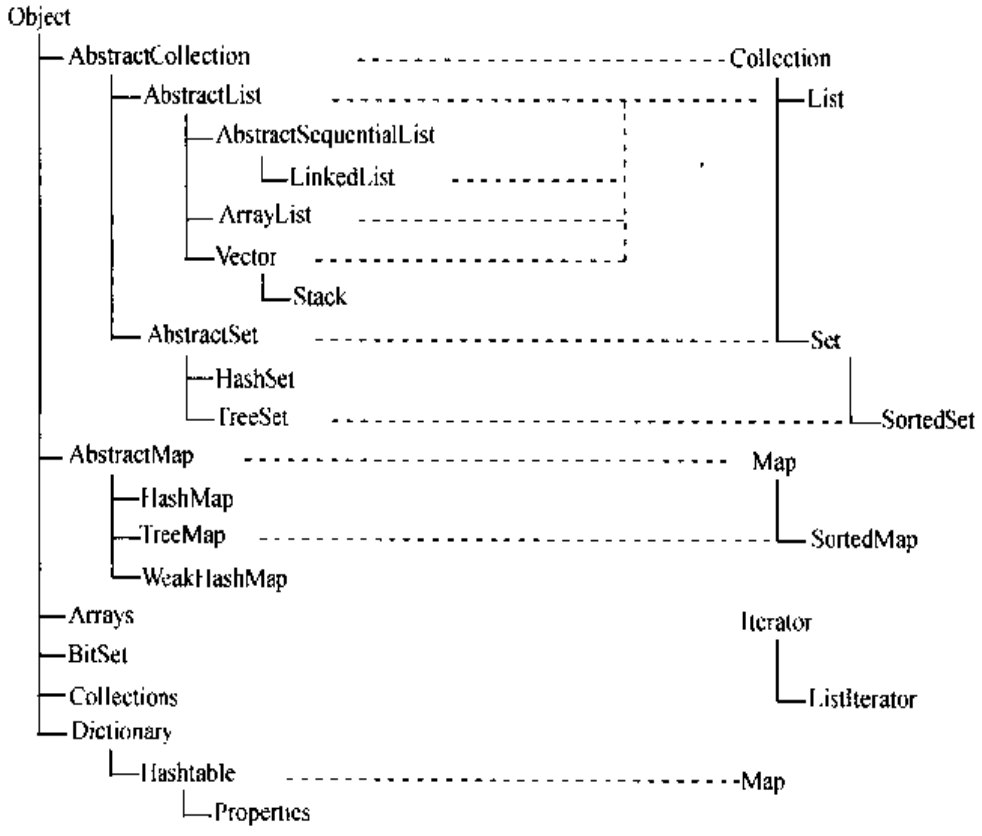


图 5.1　汇集接口、实现类之间的相互关系

图 5.1 中的虚线将右边的接口和左边的直接实现该接口的类连接起来。例如，Hashtable 类实现了 Map 接口。对继承关系而言，不管是类之间的，还是接口之间的，均用实线表示。例如，Stack 类扩展了 Vector 类，以及 SortedSet 接口扩展了 Set 接口等。

注意 4 个主要接口 Collection、List、Set 和 Map 中的每一个接口均由一个相应的抽象类 AbstractCollection、AbstractList、AbstractSet 和 AbstractMap 实现。

同时还应注意到多数实际汇集类的名字是由所用数据结构以及所实现的接口名拼接而成的，例如，ArrayList 类就使用一个来实现 List 接口，而 HashSet 类使用哈希表来实现 Set 接口，TreeMap 类则使用树形结构来实现 Map 接口。

本章主要概述汇集接口和循环接口的直接实现，而其他 6 个接口将分别在第 7、第 13 和第 14 章讨论。

5.2 Collection 接口

正如图 5.1 所示，Collection 接口为 Java 中的所有包容类定义了一个框架，该接口位于 java.util 包中，其定义如下：

```
public interface Collection
{
  public boolean   add(Object  object);
  public boolean   addAll(Collection  colllection);
  public void      clear();
  public boolean   contains(Object  object);
  public boolean   containsAll(Collection  collection);
  public boolean   equals(Object  object );
  public int       hashCode();
  public boolean    isEmpty();
  public Iterator    iterator();
  public boolean    remove(Object  object);
  public boolean    removeAll(Collection  collection);
  public boolean    retainAll(Collection  collection);
  public int        size();
  public Object[ ]   toArray();
  public Object[ ]   toArray(Object[ ]  objects);
}
```

5.3 AbstractCollection 类

类 AbstractCollection 是接口 Collection 的部分实现，实现了在对汇集所使用

标准分享网 www.bzfxw.com 免费下载

的实际存储结构不作任何说明的情况下所能实现的所有方法，位于 java.util 包中的 AbstractCollection 类定义如下：

```
public class AbstractCollection implement Collection
{

  public boolean            add(Object  object );
  public boolean            addAll(Collection  collection);
  public void               clear();
  public boolean            contains(Object  object );
  public boolean            containsAll(Collection  collection);
  public boolean            isEmpty();
  abstract public Iterator  iterator();
  public boolean            remove(Object  object);
  public boolean            removeAll(Collection  collection);
  public boolean            retainAll(Collection  collection);
  abstract public int       size();
  public Object[ ]          toArray();
  public Object[ ]          toArray(Object[ ]  objects );
  public String             toString( );
}
```

注意该类实现了Collection接口所说明的15个方法中除equals()和 hashCode()外的其余13个方法。方法 equals() 和 hashCode()在Object类中定义（见3.4节），隐含着这两个方法将在 AbstractCollection 的扩展类中覆盖。另一方面，AbstractCollection将对Object类中定义的方法toString()进行覆盖。

类 AbstractCollection 不对 Object 类中定义的 equals() 和 hashCode()方法进行覆盖的理由是这两个方法在不知道采用何种数据结构来实现包容器元素的存储结构时是不能完全定义的，而类AbstractCollection对Object类中定义的方法toString()进行覆盖的理由是它能对该覆盖方法进行定义。覆盖代码如下例所示。

例5.1 定义在AbstractCollection类中的toString()方法

```
public String toString()
{

  if ( isEmpty( ) ) return " [  ] ";
  Iterator it = iterator( ) ;
  String str = " [" + it.next ( ) ;
  while ( it. hasNext( ) )
  str += ", " + it.next( );
  return  str + " ] " ;
}
```

该方法使用了该类中的 iterator()方法所返回的循环迭代器，用于对汇集的遍历，逐渐构成所有元素的字符串影像。为了该方法能在 AbstractCollection 类的任何具体子类的对象上进行工作，它仅需要在子类中正确实现 iterator () 方法。这也是为什么在 AbstractCollection 类中将 iterator()方法定义成抽象方法的原因，它需要在具体子类中进行实现。

例 5.2 定义在 AbstractCollection 类中的 isEmpty()方法

```
public boolean isEmpty( )
{
  return  size ( ) = = 0 ;
}
```

5.4 Bag 类

一个 Bag 类(也称为多重集合) 是可包含重复元素的汇集（通常的集合中不允许包含重复元素)。在下面的例子中，我们用所有对象数组来实现 Bag 类。

例 5.3 一个 Bag 类

```
public class Bag extends AbstractCollection
{ private Object[] objects;
  private int size=0;  // number of objects in the bag
  private static final int CAPACITY=16;  // default capacity
  private void resize(int capacity)
  // increases the length of objects[] to the given capacity

  public Bag()
  { // constructs an empty bag with the default capacity
    objects = new Object[CAPACITY];
  }

  public Bag(int capacity)
  // constructs an empty bag with the given capacity

  public Bag(Collection collection)
  // constructs a bag containing the objects in the
  // given collection and with twice the capacity

  public Bag(Object[] objects)
  // constructs a bag containing the objects in the
  // given array and with twice the capacity
```

标准分享网 www.bzfxw.com 免费下载

```
public boolean add(Object object)
// adds the given object to this bag

public boolean addAll(Collection collection)
// adds all the objects in the given collection
// to this bag

public void clear()
// removes all the objects from this bag

public boolean contains(Object object)
// returns true iff the given object
// is equal to some object in this bag

public boolean containsAll(Collection collection)
// returns true iff every object in the given collection
// is equal to some object in this bag; that is,
// iff the the given collection is a subset of this bag;

private static int frequency(Collection x, Object object)
// returns the number of objects in the given collection
// that are equal to the given object

public boolean equals(Object object)
// returns true iff the given object is a Bag
// and has the same content as this bag

public int hashCode()
// returns a hash code for this bag that is the sum
// of the hash codes of its elements

public boolean isEmpty()
{ // returns true iff this bag is empty
  return size == 0;
}

public Iterator iterator()
// returns an iterator on this bag

public boolean remove(Object object)
// removes one of the given object from this bag;
// returns true iff this bag was modified;

public boolean removeAll(Object object)
```

```
  // removes all of the given object from this bag;
  // returns true iff this bag was modified;

  public boolean removeAll(Collection collection)
  // removes from this bag all of the object that are also in
  // the given collection, thereby reducing this bag to its
  // set-theoretic intersection with that collection;
  // returns true iff this bag was modified;

  public boolean retainAll(Collection collection)
  // removes from this bag all of the object that are not in
  // the given collection, thereby reducing this bag to its
  // set-theoretic complement relative to that collection;
  // returns true iff this bag was modified;

  public int size()
  { // returns the number of objects in this bag;
    return size;
  }

  public Object[] toArray()
  // returns an array whose elements are in this bag;

  public Object[] toArray(Object[] objects)
  // returns an array whose elements are in this bag
  // if it is longer than the given object[] array;
  // otherwise, the given array is returned after removing
  // all of its elements and then loading it with the elements
  // of this bag, and padding the rest of it with nulls;

  public String toString()
  // returns a String that shows the contents of this bag
}
```

例5.4 Bag.resize(int)方法的一个实现

这是一个私有的实用方法(utility method)，当把更多的元素加入到包(bag)中时，它易于使对象数组进行扩展。

```
private void resize ( int capacity)
{ // increase the length of objects[ ] to the given capacity
  if ( capacity <= this.capacity ) return;
  Object [ ] temp = objects;
  Objects = new Object[ capacity ];
  for ( int i=0; i <size ; i++ )
```

标准分享网 www.bzfxw.com 免费下载

```
    Objects[ i ] = temp [ i ];
  }
```

当给定的 capacity 不大于该对象的当前值时，该方法将不作任何工作而返回，否则，它将 objects 数组完整地拷贝到一个临时数组 temp 中，生成一个长度等于给定值 capacity 的新数组，并将该数组的引用赋给 objects，然后，将 temp 数组中的元素拷回到objects 所标识的新数组。

例 5.5 一个 Bag 构造器的实现

```
public Bag(Collection collection)
{ // constructs a bag containing the objects in the
  // given collection and with twice the capacity
  objects = new Object[2*collection.size()];
  for (Iterator it = collection.iterator(); it.hasNext(); )
    objects[size++] = it.next();
}
```

为产生可包含给定汇集对象的一个 Bag 对象，上述方法首先分配一个对象数组，该数组的大小为给定汇集对象中成份个数的两倍（留有增长的空间），然后，它用一个循环迭代器来遍历给定的汇集对象，将其对象引用变量拷贝到新包对象的对象数组中。注意此时的成份对象还没有重复，只用了一个新的数据结构来保存指向这些成份对象的引用。

因为给定的汇集对象实现了 Collection 接口，调用 iterator ()方法就可产生一个可遍历该汇集对象的循环迭代器对象，而迭代器对象实现了 java.util.Iterator 接口（见 5.5 节），就肯定可以用已在这里使用到的 next() 和 hasNext()方法，这就允许我们使用 for 循环的标准形式来遍历汇集对象，通过 next () 方法来访问它的每个元素。

例 5.6 Bag.add(Object)方法的实现

```
public boolean add( Object object )
{  // adds the given object to this bag
  if (size = = objects.length ) resize ( 2*objects.length );
  objects[ size ++] = object;
  return true;
}
```

当包对象已满时，首先调用 private resize () 方法将 objects[]的容量加倍，然后将给定的对象添加到数组中已有元素序列的后面。

注意 Collection 接口需要 add(object)方法返回布尔值来表明添加操作是否成功，使用上述实现，添加操作总是成功的（除非内存已用完），因此，它自动返回真。但我们也可见到其他的实现方式，那时 add(object)方法可以返回假。

下面是测试 add(object)方法的测试驱动程序。

```
public class Testing
{ public static void main(String[] args)
  { String[] food = { "egg", "ham", "rum", "tea" };
    Bag foodBag = new Bag(food);
    System.out.println(foodBag);
    foodBag.add("fig");
    System.out.println(foodBag);
  }
}
```

上述方法的运行结果是：

```
{ egg , ham , rum , tea }
{ egg , ham , rum , tea , fig }
```

注意，上述程序中也对 Bag (Object []) 构造器进行了测试。

例5.7 Bag.contains(Object)方法的一个实现

该方法实现了从包中查找指定对象的顺序查找算法（见2.4节），它通过对objects[]数组的查找，一旦找到数组中的某个元素等于给定的对象时，就返回真。当检查完数组中的每一元素后循环才终止，此时方法返回假。

```
public boolean contains ( Object object )
{ // 当且仅当给定对象与bag中的某个对象相等时，返回真
  for ( int i = 0; i < size ; i ++ )
    if ( object.equals( objects [ i ] ) ) return true;
  return false;
}
```

注意该方法使用了 Object.equals()方法来测试是否相等。Object.equals()只有在object与objects[i]这两个引用指向相同的对象时它才返回真值。

下面是contains (Object)方法的测试驱动程序：

```
public class Testing
   { public static void main(String[] args)
     { String[] food = { "egg", "ham", "rum", "tea" };
       Bag foodBag = new Bag(food);
       System.out.println(foodBag);
   if (foodBag.contains("fig")) System.out.println("fig");
   else System.out.println("no fig");
   if (foodBag.contains("ham")) System.out.println("ham");
   else System.out.println("no ham");
  }
}
```

标准分享网 www.bzfxw.com 免费下载

上述程序的输出为：

```
{ egg, ham, rum, tea}
not fig
ham
```

注意字符串文字的两次出现必须指向内存中的同一对象。

例5.8 Bag.equals(Object)方法的一个实现

```
public boolean equals(Object object)
{ // 当仅当给定对象为Bag类的对象，且其内容与本bag对象相同时返回真
  if (object == this) return true;
  if (object.getClass() != this.getClass()) return false;
  if (object.hashCode() != this.hashCode()) return false;
  Collection collection = (Collection)object;
  if (collection.size() != this.size()) return false;
  if (!collection.containsAll(this)) return false;
  if (!this.containsAll(collection)) return false;
  for (int i=0; i<size; i++)
  { Object x = objects[i];
    if (frequency(collection,x) != frequency(this,x))
      return false;
  }
  return true;
}
```

例5.9 Bag.removeAll(Object)方法的一个实现

```
public boolean removeAll(Object object)
 { // 从该bag中移去所有的指定对象，当仅当该bag被修改时才返回真
   boolean modified=false;
   for (int i=0; i<size; i++)
    if (object.equals(objects[i]))
    { objects[i] = objects[--size];
      modified = true;
    }
   return modified;
}
```

例5.10 Bag.toArray()方法的一个实现

```
public Object[] toArray()
{ // 返回一个包含bag中所有元素的数值
   Object[] objects = new Object[size];
for (int i=0; i<size; i++)
```

```
    objects[i] = this.objects[i];
  return objects;
}
```

例5.11 测试Bag类

下面是对定义在例子5.3中的Bag类的一个测试驱动程序：

```
import java.util.Iterator;


public class Ex0511
{ public static void main(String[] args)
{ String[] states = { "Maine", "Idaho", "Iowa", "Ohio", "Utah" };
  Bag myCollection = new Bag(states);
  print(myCollection);
  myCollection.add("Alaska");
  print(myCollection);
  if (myCollection.remove("Ohio")) print(myCollection);
  else System.out.println("Object \"Ohio\" was not found.");
  Iterator it = myCollection.iterator();
  while (it.hasNext())
  { String s = (String)it.next();
    System.out.println("\ts = \""+s+"\"");
    if (s.charAt(0) == 'I')
    { it.remove();
      System.out.println("\tObject \""+s+"\" removed.");
    }
  }
  print(myCollection);
}
private static void print(Bag collection)
{ System.out.println("size() = " + collection.size());
  Object[] objects = collection.toArray();
  for (int i=0; i<objects.length; i++)
    System.out.println("\tobjects[" + i + "] = " + objects[i]);
  if (collection.contains("Iowa"))
    System.out.println("\tContains \"Iowa\"");
  else System.out.println("\tDoes not contains \"Iowa\"");
  String[] states = { "Maine", "Idaho", "Iowa", "Ohio", "Utah" };
  Bag newCollection = new Bag(states);
  if (newCollection.containsAll(collection))
    System.out.println("\tnewC contains myC");
  else System.out.println("\tnewC does not contain myC");
  if (collection.containsAll(newCollection))
```

标准分享网 www.bzfxw.com 免费下载

```
      System.out.println("\tmyC contains newC");
    else System.out.println("\tmyC does not contain newC");
    if (collection.equals(newCollection))
      System.out.println("\tmyC equals newC");
    else System.out.println("\tmyC does not equal newC");
    if (newCollection.equals(collection))
      System.out.println("\tnewC equals myC");
    else System.out.println("\tnewC does not equal myC");
  }
}
```

上述程序的输出为：

```
size ( ) = 5
        objects[ 0] = Maine
        objects[ 1] = Idaho
        objects[ 2] = Iowa
        objects[ 3] = Ohio
        objects[ 4] = Utah
        Contains " Iowa "
        newC contains myC
        myC contains newC
        newC contains myC
size ( ) = 6
        objects[ 0] = Maine
        objects[ 1] = Idaho
        objects[ 2] = Iowa
        objects[ 3] = Ohio
        objects[ 4] = Utah
        objects[ 5] = Alaska
        Contains " Iowa "
        newC does not contains myC
        myC contains newC
        myC does not equal newC
        newC does not equal myC
size ( ) = 5
        objects[ 0] = Maine
        objects[ 1] = Idaho
        objects[ 2] = Iowa
        objects[ 3] = Ohio
        objects[ 4] = Utah
        Contains " Iowa "
        newC does not contains myC
        myC does not contains newC
        myC does not equal newC
```

```
        newC does not equal myC
        s = "Maine"
        s = "Idaho"
        Object "Idaho" removed.
        s = "Utah"
        s = "Iowa"
        Object "Iowa" removed.
        s = "Alaska"
size ( ) = 3
        objects[ 0] = Maine
        objects[ 1] = Utah
        objects[ 2] = Alaska
        Does not Contains " Iowa "
        newC does not contains myC
        myC does not contains newC
        myC does not equal newC
        newC does not equal myC
```

5.5 Iterator 接口

一个 Iterator 接口是一个遍历某个数据结构的对象，对数据结构中的每个成员刚好访问一次。由于时间关系是线性的，一个迭代器（iterator）在遍历该数据结构时也明显地将该结构进行了线性化，一个结构可以同时有多个迭代器对它进行遍历，在任何时候，每个迭代器标识结构中的单个元素，允许对该元素进行访问。

下面是一个定义在 java.util 包中的 Iterator 接口。

```
public interface Iterator
{ public boolean hasNext( );
  public Object next( );
  public void remove ( ) ;
}
```

迭代器通常以一个匿名内部类的方式加以实现，并在相关汇集类的 iterator()中进行定义，如下例所示。

例 5.12 用于 bag 类的 iterator()方法

下面是 Bag 类中 iterator()方法的实现（该 Bag 类在例子 5.3 中定义）。

```
public Iterator iterator( )
{// returns an iterator on this bag
  return  new Iterator( )  // anonymous inner class
  {  private int cursor = 0;
     public boolean hasNext( );
```

标准分享网 www.bzfxw.com 免费下载

```
        { return cursor < size;
        }
        public Object next( )
         { if (cursor >= size ) return null;
           return objects[ cursor ++ ];
          }
         public void remove( )
         { objects[ -- cursor] = objects[ - - size ];
           objects[ size ] = null;
          }
    }
}
```

复 习 题

5.1 什么是Java汇集框架?

5.2 为什么定义在AbstractCollection类中的iterator()和size()方法要声明成抽象方法?

5.3 迭代器是什么?

习 题

5.1 实现用于Bag类的以下构造函数(见例子5.3)。

```
public Bag( int capacity)
// 用给定的capacity值，构造一个空包对象
```

5.2 实现用于Bag类的以下构造函数(见例子5.3)。

```
public Bag( Object [ ] objects )
// 构造一个包含指定数组中的对象的包对象，并使包的大小为数组大小的两倍
```

5.3 为Bag类实现以下方法(见例子5.3)。

```
public boolean addAll( Collection collection)
// 将汇集中的所有对象加入到包对象中
```

5.4 为Bag类实现以下方法(见例子5.3)。

```
public void clear( )
// 移出包中的所有对象
```

5.5 为Bag类实现以下方法(见例子5.3)。

```
public boolean containsAll(Collection collection )
```

```
// 当仅当collection的每个对象与包中某个对象相等时才返回真。也就是说，当仅当
// collection为包的子集时才返回真
```

5.6 为Bag类实现以下方法（见例子5.3）。

```
private static int frequency(Collection x, Object object )
// 返回x中与对象object相等的对象的个数
```

5.7 为Bag类实现以下方法（见例子5.3）。

```
public int hashCode( )
// 返回一个哈希值，该值为包中所有元素的哈希值之和
```

5.8 为Bag类实现以下方法（见例子5.3）。

```
public boolean remove(Object object )
// 从包中移出一个对象object，当且仅当包被修改时才返回真
```

5.9 为Bag类实现以下方法（见例子5.3）。

```
public boolean removeAll(Collection collection )
// 从包中移出所有collection中的对象，也就是说，移出包与collection的交集。当且仅当
// 包被修改时才返回真
```

5.10 为Bag类实现以下方法（见例子5.3）。

```
public boolean retainAll(Collection collection )
// 从包中移出所有不在collection中的对象，也就是说，移出包相对于collection的补集
// 当且当包被修改时才返回真
```

5.11 为Bag类实现以下方法（见例子5.3）。

```
public object[ ] toArray(Object[ ] objects )
// 当包的尺寸比给定的数组大小还要大时将返回一个数组，该数组的元素为包中的元素。否则，
// 将数组中的元素全部移出，将包中的元素重新装入数组，并用空值填充数组中的多余位置
```

5.12 为Bag类实现以下方法（见例子5.3）。

```
public String toString( )
// 返回一个显示包内容的字符串
```

复习题答案

5.1 Java汇集框架是定义在java.util包中的一组接口和类，通过继承它们将有助其他数据结构的定义。

5.2 将定义在AbstractCollection类中的iterator()和size()方法声明成抽象方法是因为它们将由其他方法所使用，如toString()和isEmpty()方法，这些方法均在AbstractCollection类中加以实现，这样就可保证这些方法可在具体的子类中正常工作。

标准分享网 www.bzfxw.com 免费下载

5.3 迭代器是一个对象，它按顺序方式从汇集结构中的一个元素到达下一元素，提供了访问汇集元素的一种方法。

习 题 解 答

5.1 为 Bag 类实现 **Bag(int)** 构造函数：

```
public Bag(int capacity)
{ // constructs an empty bag with the given capacity
  objects = new Object[capacity];
}
```

5.2 为 Bag 类实现 **Bag(object [])** 构造函数：

```
public Bag(Object[] objects)
{ // constructs a bag containing the objects in the
  // given array and with twice the capacity
  this.objects = new Object[2*objects.length];
  for (int i=0; i<objects.length; i++)
    this.objects[size++] = objects[i];
}
```

5.3 为 Bag 类实现 **addAll(Collection)** 方法：

```
public boolean addAll(Collection collection)
{ // adds all the objects in the given collection
  // to this bag
  resize(2*collection.size());
  for (Iterator it = collection.iterator(); it.hasNext(); )
    objects[size++] = it.next();
  return true;
}
```

5.4 为 Bag 类实现 **clear()** 方法：

```
public void clear()
{ // removes all the objects from this bag
  for (int i=0; i<size; i++)
    objects[i] = null;
  size = 0;
}
```

5.5 为 Bag 类实现 **containsAll(Collection)** 方法：

```
public boolean containsAll(Collection collection)
{ // returns true iff every object in the given collection
```

```
  // is equal to some object in this bag; that is,
  // iff the the given collection is a subset of this bag;
  for (Iterator it = collection.iterator(); it.hasNext(); )
    if (!this.contains(it.next())) return false;
  return true;
}
```

5.6 为Bag类实现**frequency(Collection , Object)**方法：

```
private static int frequency(Collection x, Object object)
{ // returns the number of objects in the given collection
  // that are equal to the given object
  int count=0;
  for (Iterator it = x.iterator(); it.hasNext(); )
    if (object.equals(it.next())) ++count;
  return count;
}
```

5.7 为Bag类实现**hashCode()**方法：

```
public int hashCode()
{ // returns a hash code for this bag that is the sum
  // of the hash codes of its elements
  int code=0;
  for (int i=0; i<size; i++)
    code += objects[i].hashCode();
  return code;
}
```

5.8 为Bag类实现**remove(Object)**方法：

```
public boolean remove(Object object)
{ // removes one of the given object from this bag;
  // returns true iff this bag was modified;
  for (int i=0; i<size; i++)
    if (object.equals(objects[i]))
    { objects[i] = objects[--size];
      return true;
    }
  return false;
}
```

5.9 为Bag类实现**removeAll(Collection)**方法：

```
public boolean removeAll(Collection collection)
{ // removes from this bag all of the object that are also in
  // the given collection, thereby reducing this bag to its
```

标准分享网 www.bzfxw.com 免费下载

```
 // set-theoretic intersection with that collection;
 // returns true iff this bag was modified;
 boolean modified=false;
 for (Iterator it = collection.iterator(); it.hasNext(); )
  if (this.remove(it.next())) modified = true;
 return modified;
}
```

5.10 为 Bag 类实现 **retainAll(Collection)** 方法：

```
public boolean retainAll(Collection collection)
{ // removes from this bag all of the object that are not in
 // the given collection, thereby reducing this bag to its
 // set-theoretic complement relative to that collection;
 // returns true iff this bag was modified;
 boolean modified=false;
 for (int i=0; i<size; i++)
  if (!collection.contains(objects[i]))
  { remove(objects[i]);
   modified = true;
  }
 return modified;
}
```

5.11 为 Bag 类实现 **toArray(object[])** 方法：

```
public Object[] toArray(Object[] objects)
{ // returns an array whose elements are in this bag
  // if it is longer than the given object[] array;
  // otherwise, the given array is returned after removing
  // all of its elements and then loading it with the elements
  // of this bag, and padding the rest of it with nulls;
  if (size > objects.length) objects = this.toArray();
  else
  { for (int i=0; i<size; i++)
     objects[i] = this.objects[i];
    for (int i=size; i<objects.length; i++)
     objects[i] = null;
  }
  return objects;
}
```

5.12 为 Bag 类实现 **toString()** 方法：

```
public String toString()
{ // returns a String that shows the contents of this bag
  String s="{ ";
```

```
    if (size>0) s += this.objects[0];
    for (int i=1; i<size; i++)
      s += ", " + this.objects[i];
    return s + " }";
  }
```

标准分享网 www.bzfxw.com 免费下载

第6章 栈

栈(stack)是实现后进先出协议的一种包容器，这意味着包容器中惟一可访问的对象是插入到栈中的最后一个对象，可以用一堆书作一个很好的类比，当我们没有把压在某本书上的其他所有的书移走时，我们不能拿到这本书，见图6.1。



图6.1 一堆书

6.1 Java中的Stack类

如5.1节中的图5.1所示，Java集合框架中包含了一个Stack类,这个定义在java.util包中的类如图6.2所示:

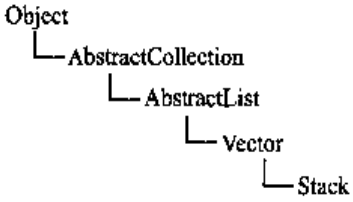


图6.2 Stack及其父类

```
public class Stack extends Vector
{
  public boolean empty( )
   { return size( ) == 0 ;
   }
  public Object peek( )
   {
     if ( size( ) = = 0 ) throw new EmptyStackException ( ) ;
     return elementAt( size ( ) -1 );
   }
  public Object pop( )
   {
     Object object = peek( );
     removeElementAt( size ( ) -1 );
     return  cbject;
   }
   public Object  push(Object object )
   {
    addElement(object );
    return  object ;
   }
public int search(Object  object )
   {
   int  i=lastIndexof(object );
```

```
    if ( i < 0) return  -1 ;  // object is not on this stack
    return  size( ) -i ;
  }
 public Stack( )
  {
  }
```

Stack 类是 Vector 类的一个子类（见第 2 章），它以一个 Vector 的形式来实现 Stack，把向量中的最后一个元素当成栈顶元素。

方法 empty() 当且仅当栈中元素个数为 0 时返回 true。

方法 peek()将栈顶元素返回，但并不将该元素从栈中移去，它使用 Vector.elementAt(int)方法来访问向量中最后的那个元素，当栈为空时，将抛出 EmptyStackException 异常。

方法 pop()在将栈顶元素从栈中移出后，将该栈顶元素返回，它使用 Vector.removeElementAt(int)方法将栈定元素移出。请注意因为该方法调用了 peek ()方法，所以，在栈为空时，它也产生 EmptyStackException 异常。

方法 push(Object)将给定元素插入栈顶，它使用了 Vector.addElement(Object)方法，它将在向量末尾加入该对象。

方法 search(Object) 返回给定对象在栈中的位置，当给定对象不在栈中时，它返回-1。该方法调用了 Vector.lastIndexOf(Object)方法，它使用 equals()方法对向量中的每个对象与给定对象进行比较。当向量中包含若干个与给定对象相等的对象时，将返回最后一个所在的位置，这是一个与栈顶最为接近的对象，使用从 1 开始的位置标识办法，栈顶元素为位置 1。

例 6.1 查看栈顶对象

```
public class Ex0601
{ public static void main(String[] args)
  { java.util.Stack stack = new java.util.Stack();
    System.out.println("stack.size() = " + stack.size());
    System.out.println("stack.peek() = " + stack.peek());
  }
}
```

该程序的输出为：

```
stack.size( ) = 0
java.util.EmptyStackException
     at java.util.Stack.peek( Stack.java :86 )
     at Ex0601.main( Ex0601.java:10 )
```

结果表明对一个空栈对象调用 peek()方法时将产生一个 EmptyStackException 异常。

标准分享网 www.bzfxw.com 免费下载

例6.2 处理EmptyStackException异常

```
import java.util.Stack;
public class Test
{ public static void main(String[] args)

  {  System.out.println("Example 6.2: Handling an EmptyStackException
exception");
    Stack stack = new Stack();
    print(stack);
  }
  private static void print(Stack stack)
  { System.out.println("Now in print(Stack):");
    System.out.println("stack.size() = " + stack.size());
    try
    { System.out.println("stack.peek() = " + stack.peek());
    }
    catch(java.util.EmptyStackException e)
    { System.out.println(e + ": The stack is empty.");
    }
  }
}
```

该程序的输出为：

```
stack.size( ) = 0
java.util.EmptyStackException : The stack is empty.
```

这里显示了处理EmptyStackException异常的一种简单方法。

例6.3 测试push()和pop()方法

```
import java.util.Stack;
public class Ex0603
{ public static void main(String[] args)
  { Stack stack = new Stack();
    stack.push("Brazil");
    stack.push("Canada");
    stack.push("France");
    stack.push("Mexico");
    stack.push("Russia");
    stack.push("Sweden");
    stack.push("Brazil");
    stack.push("Turkey");
    print(stack);
```

```
        System.out.println("stack.search(\"Brazil\") = "
                           + stack.search("Brazil"));
        System.out.println("stack.pop() = " + stack.pop());
        System.out.println("stack.pop() = " + stack.pop());
        print(stack);
        System.out.println("stack.search(\"Brazil\") = "
                           + stack.search("Brazil"));
    }
    private static void print(Stack stack)
    { System.out.println(stack);
      System.out.println("stack.size() = " + stack.size());
      try
      { System.out.println("stack.peek() = " + stack.peek());
      }
      catch(java.util.EmptyStackException e)
      { System.out.println(e + ": The stack is empty.");
      }
    }
}
```

上述程序的输出为：

```
[ Brazil, Canada, France, Mexico, Russia, Sweden, Brazil, Turkey ]
stack.size ( ) =8
stack.peek ( ) =Turkey
stack.search( "Brazil ") =2
stack.pop( ) = Turkey
stack.pop( ) = Brazil
[Brazil, Canada, France, Mexico, Russia, Sweden ]
stack.size( ) =6
stack.peek( ) = Sweden
stack.search( " Brazil ") =6
```

在将 8 个字符串对象压入栈后，上述程序调用 print(Stack)方法来显示栈信息，然后，它查找“Brazil”对象，并将“Turkey”对象从栈中弹出，然后再次调用print(Stack)方法。注意 search(Object)将返回 2，意味着栈中有“Brazil”对象的一个拷贝，它位于从栈顶数起的第二个位置（另一个拷贝位于栈底）。

然后，它调用 pop()两次，从而首先将“Turkey”并继而将“Brazil”对象从栈中移出，然后最后一次调用print(Stack)来显示栈中的 6 个元素，“Sweden”位于栈顶，而“Brazil”位于从栈顶数起的第六个元素。

注意调用 System.out.println(stack) 调用了 Vector.toString() 方法，它将返回显示元素列表的一个字符串，该字符串用方括号与其他字符相分隔。

标准分享网 www.bzfxw.com 免费下载

6.2 栈的应用

尽管栈数据结构是最为简单的数据结构之一，然而它在一些重要的应用中是极为基本的，其中有些将在下面的例子中得到说明。

如果一个算术表达式中的每一个运算符均位于操作数的后面，称这样的表示为表达式的后缀表示，也称为逆波兰表示（reverse Polish notation, or RPN）。例如，表达式3*（4+5）的后缀表示为3 4 5 + *（我们也称表达式3*（4+5）为前缀表达式）。与前缀表达式相比，后缀表示形式易于机器处理，处理后缀表达式的计数器称为RPN计数器。

例6.4 一个RPN计数器

下面的程序分析后缀表达式，并完成所指示的算术操作，它使用一个栈来保存操作数。

```
import java.util.Stack;
import java.io.*;
public class Ex0604
{ public static void main(String[] args)
  { boolean quit=false;
    String input;
    double x, y, z;
    Stack operands = new Stack();
    while (!quit)
    { input = getString("RPN> ");
      switch (input.charAt(0))
      { case 'Q':
          quit = true;
          break;
        case '+':
          y = Double.parseDouble((String)operands.peek());
          operands.pop();
          x = Double.parseDouble((String)operands.peek());
          operands.pop();
          z = x + y;
          System.out.println("\t" + x + "+" + y + " = " + z);
          operands.push(new Double(z).toString());
          break;
        case '-':
          y = Double.parseDouble((String)operands.peek());
          operands.pop();
          x = Double.parseDouble((String)operands.peek());
          operands.pop();
          z = x - y;
```

```
            System.out.println("\t" + x + "-" + y + " = " + z);
            operands.push(new Double(z).toString());
            break;
          case '*':
            y = Double.parseDouble((String)operands.peek());
            operands.pop();
            x = Double.parseDouble((String)operands.peek());
            operands.pop();
            z = x * y;
            System.out.println("\t" + x + "*" + y + " = " + z);
            operands.push(new Double(z).toString());
            break;
          case '/':
            y = Double.parseDouble((String)operands.peek());

            operands.pop();
            x = Double.parseDouble((String)operands.peek());
            operands.pop();
            z = x / y;
            System.out.println("\t" + x + "/" + y + " = " + z);
            operands.push(new Double(z).toString());
            break;
          default:
            operands.push(input);
        }
      }
    }
    private static String getString(String prompt)
    { System.out.print(prompt);
      InputStreamReader iSReader = new InputStreamReader(System.in);
      BufferedReader bReader = new BufferedReader(iSReader);
      String input="";
      try { input = bReader.readLine(); }
      catch(IOException e) { System.out.println(e); }
      return input;
    }
}
```

执行上述程序并进行以下输入：

```
RPN> 3
RPN> 4
RPN> 5
RPN> +
       4.0 + 5.0 = 9.0
```

标准分享网 www.bzfxw.com 免费下载

```
RPN> *
      3.0 * 9.0 = 27.0
RPN> 10
RPN> /
      27.0 / 10.0 = 2.7
RPN> 1
RPN> -
      2.7 - 1.0 = 1.7000000000000002
RPN> Q
```

这次运行中处理后缀表达式 345+*10/1-，它表示的前缀表达式为 3*(4+5)/10-1。对计算过程的每个中间结果均进行输出。

在 while 循环的每一迭代步，程序将打印出 PRN>提示，然后读入输入字符串。如果字符串的首字符是'Q'，则程序将终止。假如是'+'，'-'，'*'或'/'，则完成相应的算术操作，否则，输入字符串中被假定为算术操作数，然后它被压入到操作数栈中。

当输入的是上述 4 个运算符之一时，最后两个操作数将从操作数栈中弹出，并完成相应的算术操作，并进行打印输出，然后将操作结果压入操作数栈中。

人们多喜欢前缀表达式，而不是后缀表达式，下面的例子程序将一个给定的前缀表达式转换为后缀表达式。

例 6.5 将前缀表达式转换为后缀表达式

```
import java.util.Stack;

import java.io.*;
public class Ex0605
{ public static void main(String[] args)
  { try
    { Stack stack = new Stack();
      InputStreamReader reader = new InputStreamReader(System.in);
      StreamTokenizer tokens = new StreamTokenizer(reader);
      tokens.ordinaryChar('/');  // otherwise, this would be a comment
      tokens.eolIsSignificant(true);  // default is false
      int tokenType;
      System.out.print("Enter an infix expression: ");
      while ((tokenType=tokens.nextToken()) != StreamTokenizer.TT_EOL)
      { char ch = (char)tokenType;
        if (tokenType==StreamTokenizer.TT_NUMBER)
          System.out.print(tokens.nval + " ");
        else if (ch=='+' || ch=='-' || ch=='*' || ch=='/')
          stack.push(new Character(ch));
        else if (ch==')')
          System.out.print((Character)stack.pop()+" ");
```

```
        }
        while (!stack.empty())
          System.out.print((Character)stack.pop()+" ");
      }
      catch (Exception e)
      { System.out.println(e);
      }
    }
}
```

运行上述程序，并进行以下输入：

```
Enter an infix expression : ( 80 - 30 ) * ( 40 +50 /10 )
80.0 30.0 - 40.0 50.0 10.0 / + *
```

在程序中，输入由 StreamTokenizer 对象进行分析，而 tokens 绑定到 InputStreamReader 对象，它调用 ordinaryChar('/')方法，使得它能将除法符号'/'看成一个字符，并继而调用 eolIsSignificant(true)，使得它能使用类常量 TT_EOL 来控制分析循环，当遇到行结束符时终止该循环。

在 while 循环中的每一次迭代中，tokens 对象将从输入中获得下一个符号，将其字符表示放入到变量 ch 之中，然后根据它是否是一个操作数，4 个运算符中的一个，或是右括号而分别进行不同的处理。当为一个操作数时，它被立即打印输出，这是由于在后缀表达式中，操作数总是位于运算符之前。当为 -运算符时，它将被压入栈中。当为一个右括号字符时，栈顶运算符被弹出，并打印输出。

请注意，本程序需要输入表达式是完全括好的，同时，在减号之前需要一个空格，以便它能将其识别为减法操作而不是一个 -元的负操作。

6.3 消去递归

计算机操作系统在执行一个递归函数时，是通过使用一个栈来保存每次递归调用时的当前运行状态的，当每次从递归调用返回时，它将运行时栈的栈顶状态弹出，使得它能从上次离开的位置继续运行。由于栈是操作系统用于执行递归函数的，程序员应能重写一个递归函数，使用一个显示的栈结构而不用递归调用的想法应当是合理的。

例 6.6 汉内塔问题的迭代解

下面的程序与例 4.15 中的递归程序在功能上是相等的。

```
public class Ex0606
{ public static void main(String[] args)
  { hanoi(3,'A','B','C');  // play the game with 3 disks
```

标准分享网 www.bzfxw.com 免费下载

```
  }
  private static void hanoi(int n, char x, char y, char z)
  { java.util.Stack stack = new java.util.Stack();
    stack.push(new Quad(n,x,y,z));
    while (!stack.empty())
    { Quad quad = (Quad)stack.pop();
      n = quad.n;
      x = quad.a;
      y = quad.b;
      z = quad.c;
      if (n == 1) System.out.println
        ("Move top disk from peg " + quad.a + " to peg " + quad.c);
      else
      { stack.push(new Quad(n-1,y,x,z));
        stack.push(new Quad(1,x,y,z));
        stack.push(new Quad(n-1,x,z,y));
      }
    }
  }
}
class Quad
{ public int n;
  public char a, b, c;
  public Quad(int n, char a, char b, char c)
  { this.n = n; this.a = a; this.b = b; this.c = c;
  }
}
```

上述程序的输出为：

```
Move top disk from peg A to peg C
Move top disk from peg A to peg B
Move top disk from peg C to peg B
Move top disk from peg A to peg C
Move top disk from peg B to peg A
Move top disk from peg B to peg C
Move top disk from peg A to peg C
```

从上述可以看出，递归 hanoi(int, char, char, char)方法中的每一次递归调用均被 stack.push(Quad)方法调用所取代，而递归调用中的每次返回被 stack.pop(Quad)调用所取代。而程序中的 Quad 类用于保存包含一个整数和 3 个字符的四元组。

复 习 题

6.1 为什么说栈结构是一种后进先出的结构？

6.2 把一个栈称为下列结构时是否合理?

a. 后进后出结构

b. 先进后出结构

6.3 什么是

a. 前缀表示?

b. 中缀表示?

c. 后缀表示?

6.4 确定下面的每一个后缀表示是否为真?

a. xy + z + = xyz ++

b. xy + z – = xyz – +

c. xy –z + = xyz + –

d. xy –z – = xyz – –

习 题

6.1 跟踪下列代码，显示每次调用后栈中的内容:

```
Stack stack = new Stack( );
stack.push( new Character('A'));
stack.push( new Character('B'));
stack.push( new Character('C'));
stack.pop( );
stack.pop( );
stack.push( new Character('D'));
stack.push( new Character('E'));
stack.push( new Character('F'));
stack.pop( );
stack.push( new Character('G'));
stack.pop( );
stack.pop( );
stack.pop( );
```

6.2 将下面的每个前缀表达式转换为中缀表达式:

a. – / + * *a b c d e*

b. / – *a b* * *c* + *d e*

c. / *a* + *b* * *c* – *d e*

6.3 将习题6.2中的前缀表达式转换为后缀表达式。

6.4 将下面的每个中缀表达式转换为前缀表达式:

a. (*a* + *b*) – (*c* / (*d* + *e*))

b. *a* / ((*b* / *c*) * (*d* – *e*))

标准分享网 www.bzfxw.com 免费下载

c. $(a/(b/c))*(d-e)$

6.5 将习题6.4中的中缀表达式转换为后缀表达式。

6.6 将下面的每个后缀表达式转换为前缀表达式：

a. $ab+cd-/e+$

b. $abc+de-*-$

c. $abcde////$

6.7 将习题6.6中的每个后缀表达式转换为中缀表达式。

6.8 仅使用Stack类中的构造器、push()、peek()、pop()以及empty()等方法编写以下方法：

```
private static void reverse( Stack stack);
// 将指定栈中的内容进行逆转
```

6.9 仅使用Stack类中的构造器、push()、peek()、pop()以及empty()等方法编写以下方法：

```
private static Stack reversed( Stack stack);
// 返回一个新栈，该栈中包含了原指定栈中的内容，但内容的次序与原栈正好相反
```

6.10 仅使用Stack类中的构造器、push()、peek()、pop()以及empty()等方法编写以下方法：

```
public Object penultimate( Stack stack);
// 返回一个栈中从顶上数起的第二个元素
```

6.11 仅使用Stack类中的构造器、push()、peek()、pop()以及empty()等方法编写以下方法：

```
private Object bottom( );
// 返回栈底元素
```

6.12 仅使用Stack类中的构造器、push()、peek()、pop()以及empty()等方法编写以下方法：

```
private Object popBottom( );
// 返回栈底元素，并将该元素从栈中移出
```

6.13 使用从Vector中继承的方法，重新编写习题6.8中的reverse()方法。

6.14 使用从Vector中继承的方法，重新编写习题6.9中的reverse()方法。

6.15 使用从Vector中继承的方法，重新编写习题6.10中的penultimate()方法。

6.16 使用从Vector中继承的方法，重新编写习题6.11中的bottom()方法。

6.17 使用从Vector中继承的方法，重新编写习题6.12中的popBottom()方法。

6.18 修改例6.5，使它使用一个栈中内容为基本类型char而不是一个Object引用的栈。实现用于该目的的一个Stack类，并使用一个数组来保存char值。

复习题答案

6.1 栈之所以称为 LIFO（后进先出）结构是因为最后插入栈中的那个元素总是最先从栈中移出，LIFO 是 Last-In-First-Out（后进先出）的缩写。

6.2 *a.* 不合理。因为 LILO 结构意指 Last-In-Last-Out(后进后出)，它刚好与 Last-In-First-Out（后进先出）协议相反。

b. 合理。因为 FILO 结构意指 First-In-Last-Out(先进后出)，它与 Last-In-First-Out（后进先出）协议是一致的。

6.3 *a.* 算术表达式的前缀表示将二元运算符放在它的两个操作数前面。例如，表达式“$x+2$”将写成“$+\ x\ 2$”的前缀表达形式。数学中函数的标准写法也使用前缀表示，如 $f(x)$，sinx，等等。

b. 算术表达式的中缀表示将运算符放在两个操作数的中间，中缀表达式是算术表达式的常用形式。例如，$x+2$。

c. 算术表达式的后缀表示将运算符放在两个操作数的后面。例如，表达式“$x+2$”写成后缀表达式为“$x2+$”。数学中的阶乘函数使用后缀表达形式：$n!$。

6.4 *a.* 真。因为 $(x+y)+z=x+(y+z)$。

b. 真。因为 $(x+y)-z=x+(y-z)$。

c. 假。因为 $(x-y)+z \neq x-(y+z)$。

d. 假。因为 $(x-y)-z \neq x-(y-z)$。

习 题 解 答

6.1

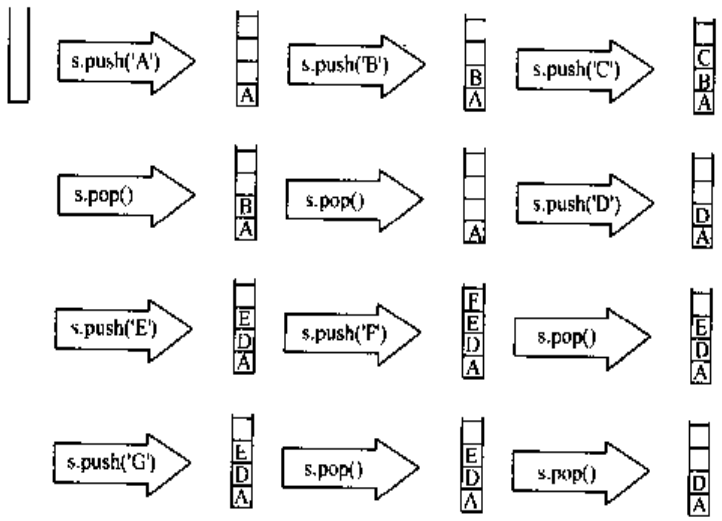


图 6.3 栈操作及其内容变化图

6.2 **a.** $(a*b+c)/d-e$

b. $(a-b)/(c*(d+e))$

标准分享网 www.bzfxw.com 免费下载

c. $a/(b+(c*(d-e)))$

6.3 a. $ab*c+d/e-$

b. $ab-cde+*/$

c. $abcde-*+/$

6.4 a. $(a+b)-(c/(d+e))=-+ab/c+de$

b. $a/((b/c)*(d-e))=/a*/bc-de$

c. $(a/(b/c))*(d-e)=*/a/bc-de$

6.5 a. $(a+b)-(c/(d+e))=ab+cde+/-$

b. $a/((b/c)*(d-e))=abc/de-*/$

c. $(a/(b/c))*(d-e)=abc//*de-*$

6.6 a. $(a+b)/(c-d)+e$

b. $a-(b+c)*(d-e)$

c. $a/(b(c/(d/e)))$

6.7 a. $+/+ab-cde$

b. $-a*+bc-de$

c. $/a/b/c/de$

6.8 将栈中内容逆转的方法：

```
private static void reverse(Stack stack)
{ Stack tempStack1 = new Stack();
  while(!stack.empty())
    tempStack1.push(stack.pop());
  Stack tempStack2 = new Stack();
  while(!tempStack1.empty())
    tempStack2.push(tempStack1.pop());
  while(!tempStack2.empty())
    stack.push(tempStack2.pop());
}
```

6.9 返回栈中逆转内容的方法：

```
private static Stack reversed(Stack stack)
{ Stack tempStack = new Stack();
  Stack newStack = new Stack();
  while(!stack.empty())
  { Object x = stack.pop();
    tempStack.push(x);
    newStack.push(x);
  }
  while(!tempStack.empty())
    stack.push(tempStack.pop());
```

```
    return newStack;
}
```

6.10 返回一个栈中从栈顶数起第二个元素的方法：

```
private static Object penultimate(Stack stack)
{ Object x1 = stack.pop();
  Object x2 = stack.pop();
  stack.push(x2);
  stack.push(x1);
  return x2;
}
```

6.11 返回栈底元素的方法：

```
private static Object bottom(Stack stack)
{ Object x = null;
  Stack tempStack = new Stack();
  while(!stack.empty())
  { x = stack.pop();
    tempStack.push(x);
  }
  while(!tempStack.empty())
    stack.push(tempStack.pop());
  return x;
}
```

6.12 返回栈底元素，并将该元素从栈中移出的方法：

```
private static Object popBottom(Stack stack)
{ Stack tempStack = new Stack();
  Object x = null;
  if (!stack.empty()) x = stack.pop();
  while(!stack.empty())
  { tempStack.push(x);
    x = stack.pop();
  }
  while(!tempStack.empty())
    stack.push(tempStack.pop());
  return x;
}
```

6.13 一个使用Vector中继承的方法，将栈中内容进行逆转的方法：

```
private static void reverse(Stack stack)
{ Stack copiedStack = (Stack)stack.clone();
  stack.clear();
```

标准分享网 www.bzfxw.com 免费下载

```
  while(!copiedStack.empty())
    stack.push(copiedStack.pop());
}
```

6.14 使用从 Vector 中继承的方法，返回栈的逆转内容的方法：

```
private static Stack reversed(Stack stack)
{ Stack copiedStack = (Stack)stack.clone();
  Stack newStack = new Stack();
  while(!copiedStack.empty())
    newStack.push(copiedStack.pop());
  return newStack;
}
```

6.15 使用从 Vector 中继承的方法，返回从顶数起第二个元素的方法：

```
private static Object penultimate(Stack stack)
{ if (stack.size()<2) return null;
  return stack.elementAt(stack.size()-2);
}
```

6.16 使用从 Vector 中继承的方法，返回栈底元素的方法：

```
private static Object bottom(Stack stack)
{ return stack.firstElement();
}
```

6.17 使用从 Vector 中继承的方法，返回栈底元素，并将栈底元素移出的方法：

```
private static Object popBottom(Stack stack)
{ return stack.remove(0);
}
```

6.18 使用一个基本类型 char 栈修改例 6.5 的程序。

```
import java.io.*;

class CharStack
{ private char[] s = new char[1000];
  private int top=-1;
  public boolean empty() { return top<0; }
  public char peek() { return s[top]; }
  public char pop() { return s[top--]; }
  public void push(char ch) { s[++top] = ch; }
}

{ public static void main(String[] args)
  { try
```

```
    { CharStack stack = new CharStack();
      InputStreamReader reader = new InputStreamReader(System.in);
      StreamTokenizer tokens = new StreamTokenizer(reader);
      tokens.ordinaryChar('/');
      tokens.eolIsSignificant(true);
      int tokenType;
      System.out.print("Enter an infix expression: ");
      while ((tokenType=tokens.nextToken())
             != StreamTokenizer.TT_EOL)
      { char ch = (char)tokenType;
        if (tokenType==StreamTokenizer.TT_NUMBER)
          System.out.print(tokens.nval + " ");
        else if (ch=='+' || ch=='-' || ch=='*' || ch=='/')
          stack.push(ch);
        else if (ch==')')
          System.out.print(stack.pop()+" ");
      }
      while (!stack.empty())
        System.out.print(stack.pop()+" ");
    }
    catch (Exception e)
    { System.out.println(e);
    }
  }
}
```

标准分享网 www.bzfxw.com 免费下载

第7章 队　列

队列是实现先进先出协议的一种容器结构，这也意味着惟一能存取到的对象为目前容器中最先插入到容器的那个对象。一个好的类比是一组正等待进入电影院的人，下一个允许进入的是排在其他人前面的那个人，如图 7.1 所示。



图 7.1　进入影院的队列

7.1　队列框架

框架是的一组接口和抽象类，这些接口和抽象类通过扩展和实现关系而相互关联，它们一起为抽象数据类型提供各种实现。Java 中的集合框架（见 5.1 节）也是用于 list、set 和 map 的一个框架。

在 Java 标准类库中，没有用于队列的框架，然而，集合框架为程序员建立队列框架提供了参考。

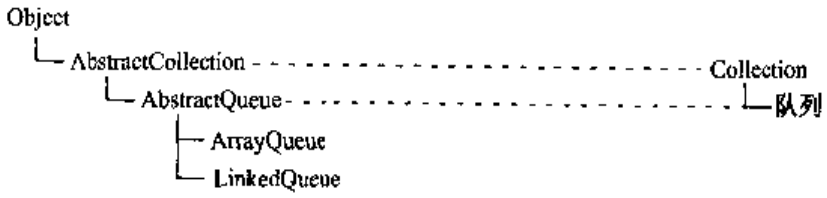


图 7.2　队列与集合框架结构

正像 List 和 Set 接口，Queue 接口应当是 Collection 接口（见本书 5.2 节）的子接口，使得它能继承以下 15 个方法说明：

```
public  interface Collection
{ public boolean   add(Object);
  public boolean   addAll(Collection);
  public void      clear();
  public boolean   contains(Object);
  public boolean   containsAll(Collection);
  public boolean   equals(Object);
  public int       hashCode();
  public boolean   isEmpty();
  public Iterator  iterator();
  public  boolean  remove(Object);
  public boolean   removeAll(Collection);
  public boolean   retainAll(Collection);
  public int       size();
  public Object[]  toArray();
```

```
  public Object[]   toArray(Cbject[]);
}
```

这样，队列与集合框架结构的关系就如图7.2所示。

例7.1 Queue接口

下面是一个队列接口，它增加了4个方法，这4个方法传统上用于定义队列的特殊行为。

```
public interface Queue extends Collection
 { public  Object dequeue();
   public  Object enqueue(Object object);
   public  Object getBack();
   public  Object getFront();
 }
```

这里的dequeue表示移去队列前头的对象，enqueue表示在队列尾部插入新对象。

例7.2 队列的使用

图7.3说明了队列是在enqueue(Object)和dequeue()方法的作用下是如何增长和收缩的，这两个方法的调用次序为：

```
q.enqueue("Amin");
q.enqueue("Buah");
q.enqueue("Chen");
q.enqueue("Diaz");
q.dequeue( );
q.dequeue( );
q.enqueue("Ford");
q.enqueue("Gore");
q.dequeue( );
```

这里，队列的前头位于左边，而末端在右边，调用getFront()将返回"Diaz"，而调用getBack()将返回"Gore"。

根据Java集合框架的样式（见5.1节），我们以抽象集合类的抽象子类形式来实现队列接口，然后，把数组和链接形式的具体实现作为该基类的子类。

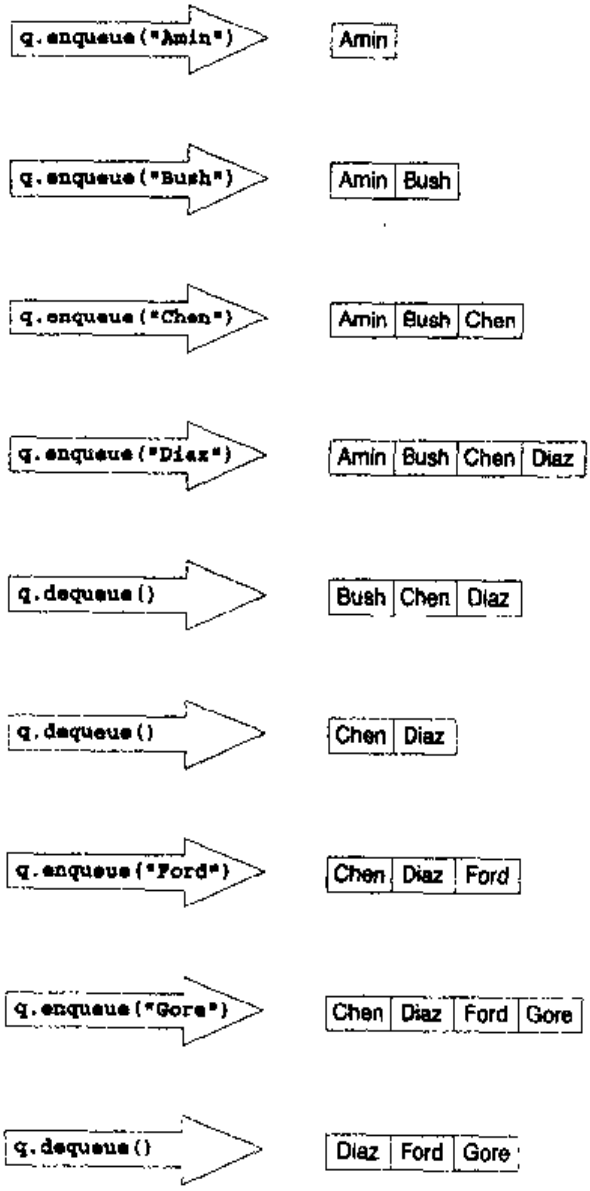


图7.3 队列操作及其内容变化图

标准分享网 www.bzfxw.com 免费下载

例7.3 AbstractQueue 类

```
import java.util.*;
import schaums.dswj.*;
public abstract
class AbstractQueue extends AbstractCollection implements Queue
{

  protected AbstractQueue()
  { // 缺省构造函数
  }

  public abstract Object dequeue();
  // to be implemented for the Queue interface

  public abstract Object enqueue(Object object);
  // to be implemented for the Queue interface

  public boolean equals(Object object)
  { // overrides the equals() method from the Object class
    if (object == this) return true;
    if (!(object instanceof AbstractQueue)) return false;
    AbstractQueue abstractQueue = (AbstractQueue) object;
    if (abstractQueue.size() != size()) return false;
    return containsAll(abstractQueue);
  }

  public abstract Object getBack();
  // to be implemented for the Queue interface

  public abstract Object getFront();
  // to be implemented for the Queue interface

  public int hashCode()
  { // overrides the hashCode() method from the Object class
    int n = 0;
    for (Iterator it = iterator(); it.hasNext(); )
    { Object object = it.next();
      if (object != null) n += object.hashCode();
    }
    return n;
  }

  public abstract Iterator iterator();
```

```
    // inherited from the java.util.AbstractCollection class

  public abstract int size();
    // inherited from the java.util.AbstractCollection class

}
```

这个抽象类实现了一个缺省构造函数，并覆盖了来自于Object类的equals(Object)和hashCode()方法，其余6个方法均声明为抽象方法，因而其实现将推迟到具体类中进行实现。方法dequeue()、enqueue(Object)、getBack()以及getFront()都是Queue接口需要的，而iterator()和size()方法是AbstractCollection所需要的。

作为AbstractCollection基类的子类，AbstractQueue类继承了以下具体方法（见5.3节）。

```
public boolean    addAll(Collection);
public void       clear();
public boolean    contains(Object);
public boolean    containsAll(Collection);
public boolean    isEmpty();
public boolean    remove(Object);
public boolean    removeAll(Collection);
public Object[]   toArray();
public Object[]   toArray(Object[]);
public String     toString( );
```

方法isEmpty()使用了size()方法，其余的10个具体方法都使用了iterator()方法，因此，任何子类都应当实现size()和iterator()方法。

7.2 顺序实现

使用数组来实现队列是最为简单的方式。

例7.4 ArrayQueue类

```
import java.util.*;
public class ArrayQueue extends AbstractQueue
{ protected Object[] objects;
  protected int front=0;
  protected int back=0;
  protected int capacity=16;
  // INVARIANTS: objects[i] == null for 0 <= i < front;
  //             objects[i] != null for front <= i < back;
```

标准分享网 www.bzfxw.com 免费下载

```
//              objects[i] == null for back <= i < capacity;

public ArrayQueue()
 { objects = new Object[capacity];
 }

 public ArrayQueue(int capacity)
 { this.capacity = capacity;
   objects = new Object[capacity];
 }

 public Object dequeue()
 { if (isEmpty()) throw new NoSuchElementException("Queue is empty");
   Object object = objects[front++];
   if (2*front>=capacity)  // shift
   { for (int i=0; i<size(); i++)
       objects[i] = objects[i+front];
     back -= front;
     front = 0;
   }
   return object;
 }

 public Object enqueue(Object object)
 { if (back>=capacity)
   { Object[] temp = objects;
     capacity *= 2;  // double the capacity
     objects = new Object[capacity];
     for (int i=0; i<back-front; i++)
       objects[i] = temp[i+front];
     back -= front;
     front = 0;
   }
   objects[back++] = object;
   return object;
 }
 public Object getBack()
 { if (isEmpty()) throw new NoSuchElementException("Queue is empty");
   return objects[back-1];
 }

 public Object getFront()
 { if (isEmpty()) throw new NoSuchElementException("Queue is empty");
   return objects[front];
```

```
    }

    public Iterator iterator()
    { return new Iterator()  // anonymous inner class
      { private int cursor=front;
        public boolean hasNext()
        { return cursor<back;
        }
        public Object next()
        { if (cursor>=back) throw new NoSuchElementException();
          return objects[cursor++];
        }
        public void remove()
        { throw new UnsupportedOperationException();
        }
      };

    }
    public int size()
    { return back-front;
    }
}
```

上述实现中的对象数组用于保存队列的元素，正如上述类不变式中所说明的那样，只有子数组objects[front .. (back -1)]是可用的，而其他元素均为空(null)。

上述类定义中有两个构造函数：一个缺省的构造函数，使得数组的最初大小为16，另一个构造函数允许使用者自己设置初始数组大小。

方法getFront()和getBack()分别返回objects[front]和objects[back -1]。方法size()返回back-front，这就是位于子数组objects[front .. (back -1)]中的元素的个数。

方法enqueue(Object)将指定的对象插入到objects[back]之中，并使数组索引back增一。当objects[back]不存在时，也就是说，back已经到达数组的尾端，此时，在插入操作之前，建立一个数组容量为原数组两倍的新数组，并把原数组中的所有back-front个元素拷贝到新数组中，在新数组中的存放位置从数组索引0开始。

方法dequeue()将位于objects [front]处的对象从队列中移出，并将front索引减1。在减1以后，如果front索引值已达到数组长度的一半时，则把剩余的back - front个元素向前进行挪动，挪动到从数组索引0开始的位置至back-front。

最后，方法iterator()返回一个遍历该队列的迭代器。由于队列以子数组的方式实现，该迭代器只需保持对数组索引的跟踪，命名为游标（cursor），它将依次通过子数组中的每个元素，返回的迭代器是由一个匿名的内部类的构造函数建立的，该类定义了Iterator接口所需的3个方法。游标被初始化为队列中的front索引值。每次对next()

标准分享网 www.bzfxw.com 免费下载

的调用均返回objects[cursor]对象，并使得cursor的值加1。在hasNext()为false（即cursor已经到达了队列尾）的情况下调用了next()，将返回NoSuchElementException异常。方法remove()将自动引发UnsupportedOperationException异常，这是因为队列不允许除dequeue()方法以外的其他方法从队列中移出元素的缘故。

例7.5 测试ArrayQueue类

```
public class TestQueue
{ public static void main(String[] args)
  { ArrayQueue q = new ArrayQueue();    System.out.println(q);
    q.enqueue("Amin");                  System.out.println(q);
    q.enqueue("Bush");                  System.out.println(q);
    q.enqueue("Chen");                  System.out.println(q);
    q.enqueue("Diaz");                  System.out.println(q);
    q.dequeue();                        System.out.println(q);
    q.dequeue();                        System.out.println(q);
    q.enqueue("Ford");                  System.out.println(q);
    q.enqueue("Gore");                  System.out.println(q);
    q.dequeue();                        System.out.println(q);
    System.out.println("q.getFront() = " + q.getFront());
    System.out.println("q.getBack() = " + q.getBack());
    System.out.println("q.size() = " + q.size());
    for (java.util.Iterator it=q.iterator(); it.hasNext(); )
      System.out.println("\tit.next() = " + it.next());
    for (;;) // force an exception
    { q.dequeue();
      System.out.println(q);
    }
  }
}
```

上述程序的运行结果为：

```
[]
[Amin]
[Amin, Bush]
[Amin, Bush, Chen]
[Amin, Bush, Chen, Diaz]
[Bush, Chen, Diaz]
[Chen, Diaz]
[Chen, Diaz, Ford]
[Chen, Diaz, Ford, Gore]
[Diaz, Ford, Gore]
q.getFront() = Diaz
```

```
q.getBack() = Gore
q.size() = 3
       it.next() = Diaz
       it.next() = Ford
       it.next() = Gore
[Ford, Gore]
[Gore]
[]
java.util.NoSuchElementException: Queue is empty
       at LinkedQueue.dequeue(LinkedQueue.java:26)
       at TestQueue.main(TestQueue.java, Compiled Code)
Exception in thread "main"
```

7.3 链接实现

尽管用数组来实现队列是最为简单的，但用链接的方式来实现队列可能更为有效。

例7.6 LinkedQueue 类

```
import java.util.*;
import AbstractQueue;
public class LinkedQueue extends AbstractQueue
{ private static class Node
  { Object object;
    Node next, previous;
    Node()
    { this.next = this.previous = this;
    }
    Node(Object object, Node next, Node previous)
    { this.object = object;
      this.next = next;
      this.previous = previous;
    }
  }
  private Node header = new Node();
  private int size = 0;

  public Object dequeue()
  { if (isEmpty()) throw new NoSuchElementException("queue is empty");
    Object object = header.next.object;
    header.next = header.next.next;
    header.next.previous = header;
    --size;
    return object;
```

标准分享网 www.bzfxw.com 免费下载

```
}

public Object enqueue(Object object)
{ Node p = header.previous;  // last element in queue
  header.previous = p.next = new Node(object,header,p);
  ++size;
  return object;
}

public Object getBack()
{ if (isEmpty()) throw new NoSuchElementException("queue is empty");
  return header.previous.object;
}

public Object getFront()
{ if (isEmpty()) throw new NoSuchElementException("queue is empty");
  return header.next.object;
}

public Iterator iterator()
{ return new Iterator()  // anonymous inner class
  { private Node cursor=header;
    public boolean hasNext()
    { return cursor.next != header;
    }
    public Object next()

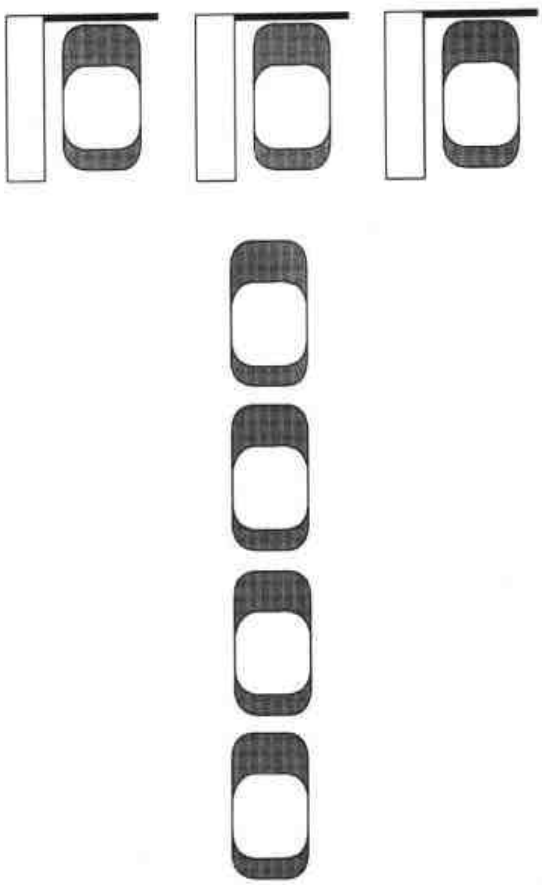
    { if (cursor.next==header) throw new NoSuchElementException();
      cursor = cursor.next;
      return cursor.object;
    }
    public void remove()
    { throw new UnsupportedOperationException();
    }
  };
}

public LinkedQueue()
{
}

public int size()
{ return size;
```

```
    }
}
```

7.4 队列应用



在客户/服务关系中，当遇到请求速度超过服务所能提供的速度时。将自然地想到用队列来进行模拟实现。例如，当小车到达收费点时，在它能进入某个服务口之前它往往需要排队等待，图 7.4 中显示了有 4 辆小车等在一个有 3 个服务口的露天汽车停车场的情景。这样的情景也将出现在一个共享少数几台打印机的局部网中，打印作业将放入一个打印队列中。超市的出口和理发店也会有类似的情况。

图 7.4 停车场排列服务

例 7.7 模拟客户/服务系统

下面的这个程序模拟的是一般客户/服务系统。客户可以是小车，打印作业或人，而相应的服务可以是收费、打印机、理发。

下面给出该程序的一个运行实例中所产生的输出的第一部分，内容如下：

```
Job #1 arrives at time 2 with 7 pages.
The queue now contains 1 job:[#1(7)]
Printer A(89%, 84% )begins Job #1 at time 2.
The queue is now empty.
Job #2 arrives at time 10 with 39 pages.
The queue now contains 1 job:[#2(39)]
Printer B(97%, 91% )begins Job #2 at time 10.
The queue is now empty.
Printer A(89%,84%) ends Job #1 at time 11.
Job #3 arrives at time 18 with 36 pages.
The queue now contains 1 job:[#3(36)]
Printer A(89%, 87% )begins Job #3 at time 18.
The queue is now empty.
Job #4 arrives at time 44 with 126 pages.
The queue now contains 1 job:[#4(126)]
Printer C(106%,104% )begins Job #4 at time 44.
The queue is now empty.
Printer B(97%,91%) ends Job #2 at time 53
Printer A(89%,87%) ends Job #3 at time 60
```

标准分享网 www.bzfxw.com 免费下载

```
Job #5 arrives at time 78 with 170 pages.
The queue now contains 1 job:[#5(170)]
Printer A(89%, 92% )begins Job #5 at time 78.
The queue is now empty.
Job #6 arrives at time 113 with 172 pages.
The queue now contains 1 job:[#6(172)]
Printer B(97%, 95% )begins Job #6 at time 113.
The queue is now empty.
Job #7 arrives at time 121 with 40 pages.
The queue now contains 1 job:[#7(40)]
Printer D(128%,124% )begins Job #7 at time 121.
The queue is now empty.
Job #8 arrives at time 127 with 30 pages.
The queue now contains 1 job:[#8(30)]
Job #9 arrives at time 136 with 41 pages.
The queue now contains 2 jobs:[#8 (30) , #9(41)]
Job #10 arrives at time 140 with 20 pages.
The queue now contains 3 jobs:[#8 (30) , #9(41) , #10(20)]
Job #11 arrives at time 147 with 31 pages.
The queue now contains 4 jobs:[#8 (30) , #9(41) , #10(20) , #11(31)]
Printer D(128%, 124% )ends Job #7 at time 154.
Printer D(128%, 126% )begins Job #8 at time 155.
The queue now contains 3 jobs:[#9(41) , #10(20) , #11(31)].
Job #12 arrives at time 160 with 63 pages.
The queue now contains 4 jobs:[#9(41) , #10(20) , #11(31) , #12(63)]
Printer C(106%, 102% )ends Job #4 at time 168.
Printer C(106%, 104% )begins Job #9 at time 169.
The queue now contains 3 jobs:[#10(20) , #11(31) , #12(63)].
Printer D(128%, 126% )ends Job #8 at time 179.
Printer D(128%, 118%)begins Job #10 at time 180.
The queue now contains 2 jobs:[#11(31) , #12(63)]
Printer D(128%, 118% )ends Job #10 at time 197.
Printer D(128%, 138%)begins Job #11 at time 198.
The queue now contains 1 job:[#12(63)]
Printer C(106%, 104% )ends Job #9 at time 209.
Printer C(106%, 96%)begins Job #12 at time 210.
The queue is now empty.
```

这个输出是用客户为打印作业，而服务为打印机这样的场景中的惯用语来进行表述的。在此次运行中，模拟程序中有 4 个服务打印机，分别为 Printer A、Printer B、Printer C 和 Printer D。每台打印机具有不同的打印速度，以百分数表示。例如，Printer A 的打印速度为 89%，表示它每秒平均打印 0.89 页。打印机的每个作业也有一个打印速度，例如，对作业 #1 而言，Printer A 的打印速度为 84%，表明它对该作业而言，每秒平均打印 0.84 页。打印机的

打印速度随打印作业的不同而有所不同，这是由一些其他因素，如网络的速度所导致的。例如，Printer B 的打印速度为每秒平均打印 0.97 页，打印作业 #2 的速度为每秒 0.91 页，而打印作业#6 时为每秒 0.95 页。

打印作业到达打印机的时间是随机的。在本次运行中，作业#1 到达打印机的时间是 2（表示从开始后的第 2 秒钟），而作业#2 的到达时间为 10，作业#3 的到达时间为 18，作业到达时间、服务时间、打印机分配等可用图 7.5 表示。

当作业到达的速度快于打印机能开始打印它们的速度时，它们将放入打印队列中。在本次模拟中，当作业#8 在时间 127 到达时，由于所有 4 台打印机均忙，就出现了这种情况。在这之前，当打印作业到达时，至少总有有一台空闲的打印机可以用于打印到达的作业。

在时间点 2，当打印作业#1 到达时，Printer A 开始打印作业#1

在时间点 10，当打印作业#2 到达时，Printer B 开始打印作业#2

在时间点 18，当打印作业#3 到达时，Printer A 开始打印作业#3

在时间点 44，当打印作业#4 到达时，Printer C 开始打印作业#4

在时间点 78，当打印作业#5 到达时，Printer A 开始打印作业#5

在时间点 113，当打印作业#6 到达时，Printer B 开始打印作业#6

在时间点 121，当打印作业#7 到达时，Printer D 开始打印作业#7

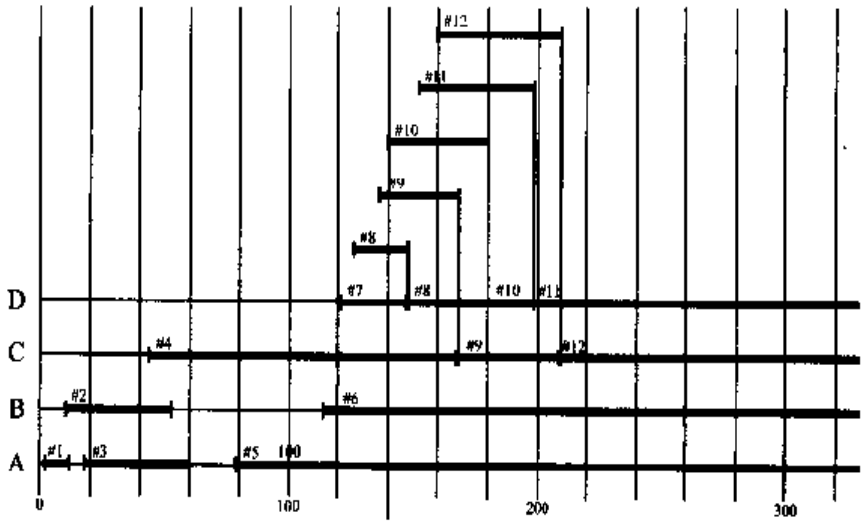


图 7.5 作业到达与服务情况图

在输出中，每个打印作业用它的 ID 号以及它的大小进行标识，例如，#1（7）表示作业#1 有 7 个页面需要打印，而#8（30）表示作业#8 有 30 个页面需要打印。

对打印作业的有效模拟需要使用随机产生的数字输入，该程序使用了 3 个随机数生成器：一个用于产生每台打印机的平均打印速度，第二个为每台打印机的各打印作业产生打印速度，第三个用于产生打印作业的到达间隙。它们是以下 Random 类的实例，而该 Random 类又是 java.util.Random 类的扩展类。

```
public class Random extends java.util.Random
```

标准分享网 www.bzfxw.com 免费下载

```
{ private double mean;
  private double standardDeviation;
  public Random(double mean)
  { this.mean = mean;
    this.standardDeviation = mean;
  }

  public Random(double mean, double standardDeviation)
  { this.mean = mean;
    this.standardDeviation = standardDeviation;
  }

  public double nextGaussian()
  { double x = super.nextGaussian();  // x = normal(0.0, 1.0)
    return x*standardDeviation + mean;
  }
  public double nextExponential()
  { return -mean*Math.log(1.0 - nextDouble());
  }

  public int intNextExponential()
  { return (int)Math.ceil(nextExponential());
  }
}
```

其中的 nextGaussian()方法将返回一个随机数，随机数按给定的平均数和标准偏差值正态分布，它调用并覆盖了 java.util.Random 类中的匿名方法，该匿名方法返回的随机数按平均数为 0.0，而标准偏差值为 1.0 进行正态分布。方法 nextExponential() 返回的随机数按给定平均数进行指数分布，这是用于随机到达间隙时间的一个比较准确的分布，它也用于产生作业的大小，而该大小将用于确定服务所需的时间。

每个打印作业是以下类的一个实例。

```
public class Client
{ public static final int MEAN_JOB_SIZE = 100;
  private static Random randomJobSize = new Random(MEAN_JOB_SIZE);
  private static int nextId = 0;
  private int id, jobSize, tArrived, tBegan, tEnded;
  private Server server;

  public Client(int time)
  { id = ++nextId;
    jobSize = randomJobSize.intNextExponential();
//   tArrived = time;
    print(id,time,jobSize);
```

```
}

public double getJobSize()
{ return jobSize;
}

public int getWaitTime()
{ return tBegan - tArrived;
}

public int getServiceTime()
{ return tEnded - tBegan;
}

public void beginService(Server server, int time)
{ this.server = server;
//  tBegan = time;
  printBegins(server,id,time);
}

public void endService(int time)
{  //tEnded = time;
  printEnds(server,id,time);
  server = null;
}

public String toString()
{ return "#" + id + "(" + (int)Math.round(jobSize) + ")";
}

private static void print(int job, int time, double size)
{ System.out.println("Job #" + job + " arrives at time " + time
    + " with " + (int)Math.round(size) + " pages.");
}

private static void printBegins(Server server, int job, int time)
{ System.out.println("Printer " + server + " begins Job #" + job
    + " at time " + time + ".");
}

private static void printEnds(Server server, int job, int time)
{ System.out.println("Printer " + server + " ends Job #" + job
    + " at time " + time + ".");
}
```

标准分享网 www.bzfxw.com 免费下载

```
}
```

随机数产生器 randomJobSize 产生指数分布的作业大小，平均页数为 100 页。它被声明为 static，这是由于一个实例就足以产生所有作业的大小。同样，static int nextId 用于所有作业的标识数。

构造函数使用 nextId 计数器来为作业设置 id 号，同时它还使用了 randomJobSize 产生器来设置作业的大小，然后它打印一行输出，宣布该作业已经到达。

方法 beginService()将打印机赋值给 server 引用，并输出一行，表明打印已经开始，同样，endService()方法先在打印一行，宣布该打印作业结束，然后把空值赋值给 server 引用。

每个打印机是以下类的一个实例。

```
public class Server
{ private static Random randomMeanServiceRate = new Random(1.00,0.20);
  private static char nextId = 'A';
  private Random randomServiceRate;
  private char id;
  private double meanServiceRate, serviceRate;
  private Client client;
  private int timeServiceEnds;

  public Server()
  { id = (char)nextId++;
    meanServiceRate = randomMeanServiceRate.nextGaussian();
    randomServiceRate = new Random(meanServiceRate,0.10);
  }

  public Client getClient()
  { return client;
  }

  public void beginServing(Client client, int time)
  { this.client = client;
    serviceRate = randomServiceRate.nextGaussian();
    client.beginService(this,time);
    int serviceTime = (int)Math.ceil(client.getJobSize()/serviceRate);
    timeServiceEnds = time + serviceTime;
  }

  public void endServing(int time)
  { client.endService(time);
    this.client = null;
  }
```

```
  public int getTimeServiceEnds()
  { return timeServiceEnds;
  }

  public boolean isFree()
  { return client == null;
  }

  public String toString()
  { int percentMeanServiceRate = (int)Math.round(100*meanServiceRate);
    int percentServiceRate = (int)Math.round(100*serviceRate);
    return id + "(" + percentMeanServiceRate + "%,"
            + percentServiceRate + "%)";
  }
}
```

随机数产生器 randomMeanServiceRate 产生一个平均数为 100.0，而标准偏差为 20.0 的正态分布的打印速度随机值，为每台打印机产生一个 meanServiceRate 值。正如本次运行所示，它为 Printer A 产生的打印速度值为 89%，它为 Printer B 产生的打印速度值为 97%，它为 Printer C 产生的打印速度值为 106%，而为 Printer D 产生的打印速度值为 128%。同样，随机数产生器 randomServiceRate 为某个打印作业产生正态分布的速度值。在所显示的运行中，它为作业#1 所产生的速度为 84%，为作业#3 所产生的速度为 87%，为作业#5 所产生的速度为 92%，这些数值来源于平均数为 89% 的正态分布（对 Printer A），而标准偏差值设为 10%。

方法 beginServing（ ）将要打印的客户作业赋值给 client 引用，并从产生器 randomServiceRate 那里获得正态分布的 serviceRate 值，然后将 beginService 消息发送给客户打印作业，接下来，赋值语句 int serviceTime=(int)Math.ceil(client.getJobSize()/serviceRate);完成打印作业所需的时间（秒数），该值是用作业大小（页面数）除打印速度（每秒多少页）来获得的，然后将该值加入到 Server 对象中的 timeServiceEnds 时间段中。

下面是 main 方法所在的类定义。

```
import schaums.dswj.Queue;
public class ClientServerSimulation
{ private static final int NUMBER_OF_SERVERS = 4;
  private static final double MEAN_INTERARRIVAL_TIME = 20.0;
  private static final int DURATION = 100;
  private static Server[] servers = new Server[NUMBER_OF_SERVERS];
  private static Queue clients = new ArrayQueue();
  private static Random random = new Random(MEAN_INTERARRIVAL_TIME);

  public static void main(String[] args)
```

标准分享网 www.bzfxw.com 免费下载

```
{ for (int i=0; i<NUMBER_OF_SERVERS; i++)
    servers[i] = new Server();
  int timeOfNextArrival = random.intNextExponential();
  for (int t=0; t<DURATION; t++)
  { if (t == timeOfNextArrival)
    { clients.enqueue(new Client(t));
      print(clients);
      timeOfNextArrival += random.intNextExponential();
    }
    for (int i=0; i<NUMBER_OF_SERVERS; i++)
      if (servers[i].isFree())
      { if(!clients.isEmpty())
        { servers[i].beginServing((Client)clients.dequeue(),t);
          print(clients);
        }
      }
      else if (t == servers[i].getTimeServiceEnds())
        servers[i].endServing(t);
  }
}

private static void print(Queue queue)
{ int size=queue.size();
  if (size==0) System.out.println("The queue is now empty.");
  else System.out.println("The queue now contains " + size
       + " job" + (size>1?"s: ":": ") + queue);
}
}
```

在例 7.7 中，它的主循环在每个时间间隔均迭代一次，称这样的模拟实现为时间驱动模拟，相反，如果主循环是在某个事件发生时才迭代一次的话，就称为事件驱动的模拟实现，事件可以是新作业的到达，服务的开始或服务的结束。事件驱动模拟程序通常更为简单，但要求所有的服务器具有相同的工作速度。

复 习 题

7.1 为什么把队列称为先进先出结构?

7.2 将队列称为

a. 一种 LILO（后进后出）结构是否合理?

b. 一种 FILO（先进后出）结构是否合理?

7.3 队列的链接实现相对于顺序实现而言有什么优势和不足?

习 题

7.1 跟踪下列代码，显示每次调用后队列中的内容：

```
ArrayQueue q;
q.enqueue('A');
q.enqueue('B');
q.enqueue('C');
q.dequeue( );
q.dequeue( );
q.enqueue('D');
q.enqueue('E');
q.enqueue('F');
q.dequeue( );
q.enqueue('G');
q.dequeue( );
q.dequeue( );
q.dequeue( );
```

对下列每一个问题，仅使用ArrayQueue类中的构造器、enqueue()、dequeue()以及empty()等方法为它们编写相应的方法：

7.2
```
private static void reverse(ArrayQueue queue);
// 将指定队列中的内容进行逆转
```

7.3
```
private static ArrayQueue reversed(ArrayQueue queue);
// 返回一个新队列，该队列中包含了原指定队列中的内容，但内容的次序与原队列正好相反
```

7.4
```
public static Object second (ArrayQueue queue);
// 返回一个从队列头数起的第二个元素
```

7.5
```
private static Object last( );
// 返回队列中的最后一个元素
```

7.6
```
private static Object removeLast( );
// 返回队列中的最后一个元素，并将该元素从队列中移出
```

7.7
```
public static ArrayQueue merge( ArrayQueue q1, ArrayQueue q2 );
// 将两个队列合并到一起，合并的方式为交替使用两队列中的元素，并把剩余的队列中的元素添加
// 到最后，并将合并产生的新队列返回
```

7.8 修改例7.7中的程序，使得它计算并打印下列各统计数值。

a. 系统中的平均作业数。

b. 队列中的平均作业数。

c. 系统中执行一个作业所需的平均时间。

d. 一个作业在队列中进行等待所需的平均时间。

标准分享网 www.bzfxw.com 免费下载

从理论上，上述各答案应当大致等于：

a. $L=n/(st-n)$

b. $(nL)/(st)$

c. tL

d. $(nL)/s$

其中，n 为作业的平均大小，s 为服务器数目，t 为作业平均到达间隔。例如，当 n=100, $s=6, t=20$ 时，相应的答案应当近似等于：

a. 5 个作业。

b. 4.17 个作业。

c. 100 秒。

d. 83.3 秒。

7.9 修改例 7.8 中的程序，使得它计算并打印下列各统计数值。

a. 每个服务器的平均服务时间。

b. 所有作业的平均服务时间。

c. 每个服务器的空闲时间所占的百分比值。

复习题答案

7.1 队列被称为先进先出结构是因为最先插入队列中的元素总是那个最先被移出的元素。FIFO 是 First-In-First-Out(先进先出)的缩写。

7.2 *a.* 是合理的。因为后进后出结构与先进先出协议是一致的。

b. 不合理。因为先进后出结构与先进先出协议相反。

7.3 队列的链接实现是它基本上消除了队列溢出现象，对 enqueue()函数调用的次数仅受 new 操作所能获得的内存的限制。其惟一的不足是链接实现使用了指针，因此，它比顺序实现更为复杂。

习 题 解 答

7.1 见图 7.6。

7.2 将队列中的内容进行逆转的方法：

```
private static void reverse(ArrayQueue queue)
{ Stack tempStack = new Stack();
  while(!queue.isEmpty())
    tempStack.push(queue.dequeue());
  while(!tempStack.empty())
    queue.enqueue(tempStack.pop());
}
```

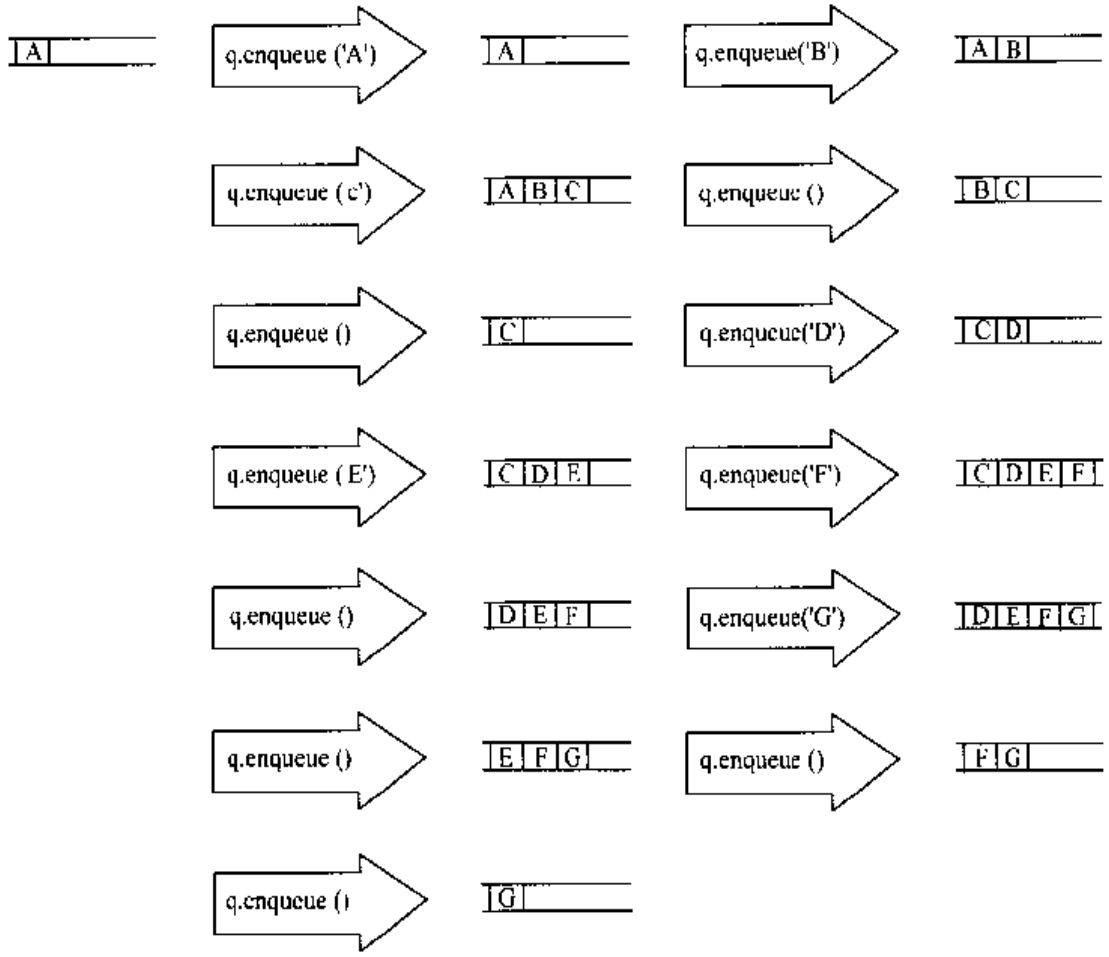


图 7.6 队列操作及内容变化跟踪图

7.3 返回队列的逆转内容的方法：

```
private static ArrayQueue reversed(ArrayQueue queue)
{ ArrayQueue newQueue = new ArrayQueue();
  Stack stack = new Stack();
  while(!queue.isEmpty())
  { Object x = queue.dequeue();
    stack.push(x);
    newQueue.enqueue(x);
  }
  while(!newQueue.isEmpty())
    queue.enqueue(newQueue.dequeue());
  while(!stack.empty())
    newQueue.enqueue(stack.pop());
  return newQueue;
}
```

7.4 返回从队列头数起第二个元素的方法：

```
private static Object second(ArrayQueue queue)
{ ArrayQueue tempQueue = new ArrayQueue();
```

标准分享网 www.bzfxw.com 免费下载

```
  Object x = queue.dequeue();
  tempQueue.enqueue(x);
  x = queue.dequeue();
  tempQueue.enqueue(x);
  while(!queue.isEmpty())
    tempQueue.enqueue(queue.dequeue());
  while(!tempQueue.isEmpty())
    queue.enqueue(tempQueue.dequeue());
  return x;
}
```

7.5 返回队列中的最后一个元素的方法：

```
private static Object last(ArrayQueue queue)
{ ArrayQueue tempQueue = new ArrayQueue();
  while(!queue.isEmpty())
    tempQueue.enqueue(queue.dequeue());
  Object x = null;
  while(!tempQueue.isEmpty())
  { x = tempQueue.dequeue();
    queue.enqueue(x);
  }
  return x;
}
```

7.6 返回队列中的最后一个元素，并将该元素从队列中移出的方法：

```
private static Object removeLast(ArrayQueue queue)
{ ArrayQueue tempQueue = new ArrayQueue();
  int count=0;
  while(!queue.isEmpty())
  { tempQueue.enqueue(queue.dequeue());
    ++count;
  }
  Object x = null;
  for (int i=0; i<count-1; i++)
    queue.enqueue(tempQueue.dequeue());
  return tempQueue.dequeue();
}
```

7.7 将两个队列进行合并的方法：

```
public static ArrayQueue merged(ArrayQueue q1, ArrayQueue q2)
{ ArrayQueue newQueue = new ArrayQueue();
  while(!q1.isEmpty() && !q2.isEmpty())
  { newQueue.enqueue(q1.dequeue());
    newQueue.enqueue(q2.dequeue());
```

```
    }
    while(!q1.isEmpty())
      newQueue.enqueue(q1.dequeue());
    while(!q2.isEmpty())
      newQueue.enqueue(q2.dequeue());
    return newQueue;
  }
```

7.8 修改例 7.7 中的程序，在 Client 类中增加以下代码：

```
private int id, jobSize, tArrived, tGegan, tEnded;
public Client( int time )
{ id = ++ nextId;
  jobsize = randomJobSize.iniNextExponential();
  tArrived = time;
public double getJobSize( )
{ return jobSize ;
}
public int getwaitTime( )
{
return tBegan - tArrived;
}
}
Pbulic int getServiceTime( )
{
return tEnded - tBegan;
}
public void beginService(Server server, int time)
{ this.server = server;
  tBegan = time;
}
public void endService( int time)
{
tEnded = time;
server = null;
}
```

在 Server 类中增加以下代码：

```
public Client getClient
{ return client;
}
```

在 main 类中增加以下代码：

```
import schaums.dswj.Queue;
public class Pr0708
```

标准分享网 www.bzfxw.com 免费下载

```
{ private static final int NUMBER_OF_SERVERS = 4;
  private static final double MEAN_INTERARRIVAL_TIME = 20.0;
  private static final int DURATION = 100;
  private static Server[] servers = new Server[NUMBER_OF_SERVERS];
  private static Queue clients = new ArrayQueue();
  private static Random random = new Random(MEAN_INTERARRIVAL_TIME);
  private static int totalTime, queueTime, totalNumberOfJobs;
  private static int[] numberOfJobs = new int[NUMBER_OF_SERVERS];
  public static void main(String[] args)
  { for (int i=0; i<NUMBER_OF_SERVERS; i++)
      servers[i] = new Server();
    int timeOfNextArrival = random.intNextExponential();
    for (int t=0; t<DURATION; t++)
    { if (t == timeOfNextArrival)
      { clients.enqueue(new Client(t));

        timeOfNextArrival += random.intNextExponential();
      }
      for (int i=0; i<NUMBER_OF_SERVERS; i++)
      { Server server = servers[i];
        if (server.isFree())
        { if(!clients.isEmpty())
          { server.beginServing((Client)clients.dequeue(),t);
          }
        }
        else // server is serving a client
        { ++totalTime;  // counting time in system
          if (t == server.getTimeServiceEnds())
          { Client client = server.getClient();
            server.endServing(t);
            int waitTime = client.getWaitTime();
            ++numberOfJobs[i];  // count only those jobs that finish
          }
        }
      }
      totalTime += clients.size();
      queueTime += clients.size();
    }
    for (int i=0; i<NUMBER_OF_SERVERS; i++)
      totalNumberOfJobs += numberOfJobs[i];
    printStatistics();
  }
  private static void printStatistics()
  { System.out.println("Mean interarrival time: " +
```

```
                    MEAN_INTERARRIVAL_TIME);
    System.out.println("Mean service time: "
      + Client.MEAN_JOB_SIZE);
    System.out.println("Number of servers: " + NUMBER_OF_SERVERS);
    System.out.println("Average number of jobs in the system: "
      + (float)totalTime/DURATION);
    System.out.println("Average number of jobs in the queue: "
      + (float)queueTime/DURATION);
    System.out.println("Average time a job spent in the system: "
      + (float)totalTime/totalNumberOfJobs);
    System.out.println("Average time a job spent in the queue: "
      + (float)queueTime/totalNumberOfJobs);
  }
```

7.9 第二次修改例7.7中的程序，加入以下代码：

```
private static int[] serviceTime = new int[NUMBER_OF_SERVERS];
private static int[] numberOfJobs = new int[NUMBER_OF_SERVERS];
public static void main(String[] args)
{ for (int i=0; i<NUMBER_OF_SERVERS; i++)
  { servers[i] = new Server();
    serviceTime[i] = numberOfJobs[i] = 0;
  }
  int timeOfNextArrival = random.intNextExponential();
  for (int t=0; t<DURATION; t++)
  { if (t == timeOfNextArrival)
    { clients.enqueue(new Client(t));
      timeOfNextArrival += random.intNextExponential();
    }
    for (int i=0; i<NUMBER_OF_SERVERS; i++)
    { Server server = servers[i];
      if (server.isFree())
      { if(!clients.isEmpty())
        { server.beginServing((Client)clients.dequeue(),t);
        }
      }
      else // server is serving a client
      { ++totalTime;  // counting time in system
        if (t == server.getTimeServiceEnds())
        { Client client = server.getClient();
          server.endServing(t);
          int waitTime = client.getWaitTime();
          serviceTime[i] = client.getServiceTime();
          ++numberOfJobs[i];  // count only those jobs that finish
        }
```

标准分享网 www.bzfxw.com 免费下载

```
        }
      }
      totalTime += clients.size();
      queueTime += clients.size();
    }
    for (int i=0; i<NUMBER_OF_SERVERS; i++)
    { totalServiceTime += serviceTime[i];
      totalNumberOfJobs += numberOfJobs[i];
    }
    printStatistics();
  }

  private static void printStatistics()
  { System.out.println("Mean interarrival time: " +
                       MEAN_INTERARRIVAL_TIME);
    System.out.println("Mean service time: "
      + Client.MEAN_JOB_SIZE);
    System.out.println("Number of servers: " + NUMBER_OF_SERVERS);
    System.out.println("Average number of jobs in the system: "
      + (float)totalTime/DURATION);
    System.out.println("Average number of jobs in the queue: "
      + (float)queueTime/DURATION);
    System.out.println("Average time a job spent in the system: "
      + (float)totalTime/totalNumberOfJobs);
    System.out.println("Average time a job spent in the queue: "
      + (float)queueTime/totalNumberOfJobs);
    System.out.println("Total time spent by all jobs in system: "
      + totalTime);
    System.out.println("Total time spent by all jobs in queue:  "
      + queueTime);
    System.out.println("Duration:                              " + DURATION);
    System.out.println("Total number of jobs completed: "
      + totalNumberOfJobs);
    System.out.println("Average service time for server:");
    for (int i=0; i<NUMBER_OF_SERVERS; i++)
      System.out.println("\t" + servers[i] + ": "
        + (float)serviceTime[i]/numberOfJobs[i]);
    System.out.println("Average service time among all jobs: "
      + (float)totalServiceTime/totalNumberOfJobs);
    System.out.println("Percent idle time for server:");
    for (int i=0; i<NUMBER_OF_SERVERS; i++)
      System.out.println("\t" + servers[i] + ": "
        + (float)(100.0*(DURATION - serviceTime[i]))/DURATION + "%");
  }
```

第8章 线 性 表

线性表是一个可在任意某个局部对元素进行插入和删除操作的序列容器，这一点与前一章中的队列结构不同，队列是一种先进先出结构，插入只能插入到队列末尾，而删除只能删除头元素。对线性表中所有元素进行操作的时间均是一样的，也就是说，操作时间与容器的大小无关，这对需要进行随机存取的应用而言，是一种备受青睐的数据结构。一列挂车就是一个好的类比，任何一节只需简单地将它从邻接的车箱松开，使得本节车箱从原有列车中脱离出来，并将先前和本节车箱相链的前后两节车箱连接起来即可，见图示 8.1。

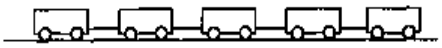


图 8.1 一列挂车

在 Java 中，List 接口是 Collection 接口的扩展，因此，该接口和它的实现均属于 Java 1.2 集合框架（见 5.1 节）中的一部分，其定义位于 java.util 包中。List 接口为数学中的序列概念提供了一种抽象数据类型，因此，List 在 Java 中的实现也就呈现一个特点，即该容器支持元素的索引存取。

8.1 java.util.List 接口

以下就是定义在 java.util 包中的 List 接口。

```
public interface List extends Collection
{
  public boolean        add(Object  object);
  public void           add(int index, Object  object);
  public boolean        addAll (Collection  collection);
  public boolean        addAll ( int index, Collection collection );
  public void           clear();
  public boolean        contains(Object  object);
  public boolean        containsAll(Collection  collection);
  public boolean        equals(Object  object);
  public Object         get( int index );
  public int            hashCode();
  public int            indexOf ( Object  object);
  public boolean        isEmpty();
  public Iterator       iterator();
  public int            lastIndexOf ( Object  object);
```

标准分享网 www.bzfxw.com 免费下载

```
  public ListIterator   listIterator ( );
  public ListIterator   listIterator(int index);
  public Object         remove(int index);
  public boolean        remove(Object object);
  public boolean        removeAll(Collection collection);
  public boolean        retainAll(Collection collection);
  public Object         set(int index, Object object);
  public int            size();
  public List           subList (int start, int stop);
  public Object[ ]      toArray();
  public Object[ ]      toArray(Object[ ]  object);
}
```

注意该类定义中使用了一种特别的迭代器，称为 ListIterator，它是一种双向的迭代器。

例 8.1 使用 List 接口

```
import java.util.*;
public class Ex0801
{ public static void main(String[] args)
  { String[] strings;
    strings = new String[]{"Adams","Tyler","Grant","Hayes","Nixon"};
    List list = Arrays.asList(strings);
    System.out.println("list = " + list);
    System.out.println("list.size() = " + list.size());
    System.out.println("list.getClass().getName() = "
                  + list.getClass().getName());
    System.out.println("list.get(1) = " + list.get(1));
    System.out.println("list.contains(\"Tyler\") = "
                   + list.contains("Tyler"));
    List sublist = list.subList(2,5);
    System.out.println("sublist = " + sublist);
    Object object = list.get(1);
    System.out.println("sublist.get(1) = " + sublist.get(1));
    System.out.println("sublist.contains(\"Tyler\") = "
                   + sublist.contains("Tyler"));
    list.remove(3);
  }
}
```

该程序的输出为：

```
list = [ Adams, Tyler, Grant, Hayes, Nixon ]
list.size( ) = 5
```

```
list.gerClass().getName( ) = java.util.Arrays$ArrayList
list.get(1) = Tyler
list.contains("Tyler") = true
sublist = [Grant, Hayes, Nixon]
sublist.ger(1) = Hayes
sublist.contains("Tyler") = false
Exception in thread "main" java.lang.UnsupportedOperationException
      at java.util.AbstractList.remove(AbstractList.java:174)
      at Ex0801.main(Ex0801.java:23)
```

这里使用了一个 String 型数组来构造 List 对象，方法 getClass()显示该对象是 ArrayList 类的一个实例。

方法 get()和 sublist() 均使用索引数字当作它的参数，方法调用 list.sublist(2, 5)返回一个 List 对象，该对象包含了线性表中从索引值为 2 到 4 的所有元素。

方法调用 list.remove(3) 引发 UnsupportedOperationException 异常，这是由于在 ArrayList 类中没有实现 remove()方法的缘故。

例 8.1 中引发的异常指出了一个事实，List 接口的某些实现并没有实现接口中的所有方法，当这些方法没有被实现时，就要求引发 UnsupportedOperationException 异常。

8.2 java.util.List 接口的实现

例 8.1 中用例说明了 Java 1.2 集合框架中实现了 List 接口的 ArrayList 类的使用，List 接口也被 AbstractList、LinkedList 以及 Vector 类所实现，它与集合框架的关系见图 8.2。

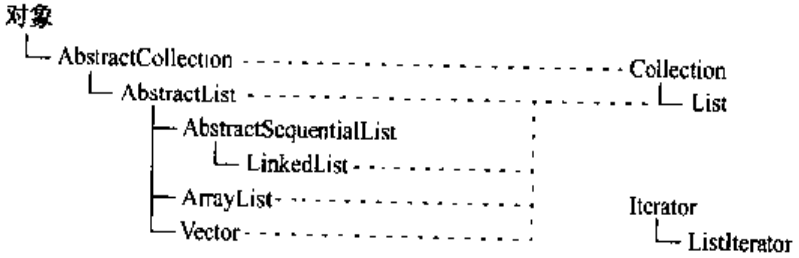


图 8.2 List 与集合框架的关系

AbstractList 类与集合中的 AbstractCollection 所起的作用是一样的，它是接口的部分实现，通过它，可减少为某个特定实现而需编写的代码量，由于它是抽象的，所以不能对它直接进行实例化，AbstractList 类将在 8.3 节中作简要说明。

LinkedList 类体现了一个线性表类中所希望的所有的结构和功能，在 8.6 节将对它进行概述。

ArrayList 类使用一个连续数组来实现 List 接口，在 8.5 节将对它进行简要说明。

Vector 类与 LinkedList 类类似，它在 2.6 节中已做过概述。

标准分享网 www.bzfxw.com 免费下载

8.3 AbstractList 与 AbstractSequentialList 类

AbstractList 类旨在为 List 类的完整实现提供一个骨架，它采用数组来存储线性表中的元素。同样，AbstractSequentialList 类也旨在为 List 类的完整实现提供一个骨架，但它采用非连续的链接结构来存储线性表中的元素。这两种不同的存储结构从底层实现的意义上讲正好相反，连续的数组结构提供了一种随机访问的直接存取方式，但在进行插入和删除操作时，需对数据进行实际的移动操作，而链接结构可以进行直接插入和删除操作，但对随机访问的直接存取方式而言，将面临没有索引的代价（需沿链接结构找到操作位置）。表 8.1 对这两种结构的不同进行了总结。

表 8.1 两种实现方式的比较

部分实现	完整实现	数据结构	Get(), set()	add(), remove()
AbstractList	ArrayList	数组	$O(1)$	$O(n)$
AbstractSequentialList	LinkedList	链接表	$O(n)$	$O(1)$

当你的应用需要在一种相当稳定的线性序列上进行快速存取是，应使用 ArrayList 类，也可对 ArrayList 或其父类 AbstractList 进行扩展。反之，如果你的应用中面临的是需要经常进行插入和删除操作的动态序列，那么，你应当采用 LinkedList，也可对 LinkedList 或其父类 AbstractSequentialList 进行扩展。

注意 LinkedList 类中并没有直接实现 get(int) 方法。

下面是定义在 java.util 包中的 AbstractList 类的结构。

```
public abstract class AbstractList extends AbstractionCollection
implements List
{ protected              AbstractList( );
  public boolean         add(Object  object);
  public void            add(int index, Object  object);
  public boolean         addAll (Collection  collection);
  public boolean         addAll ( int index, Collection collection );
  public void            clear();
  public boolean         equals(Object  object);
  abstract  public Object get( int index );
  public int             hashCode();
  public int             indexOf ( Object  object);
  public Iterator        iterator();
  public int             lastIndexOf ( Object  object);
  public ListIterator    listIterator ( );
  public ListIterator    listIterator( int index );
  public Object          remove( int index );
  public void            removeRange(int start,  int stop);
  public Object          set( int index, Object object );
```

```
    public List              subList ( int start, int stop);
}
```

其中的 get()方法被声明为抽象的，所以该类的任何具体扩展类均将实现它。方法 set()，add()以及 remove()的实现都只需简单地用以下代码来加以实现：

```
{
 throw new UnsupportedOperationException( ) ;
}
```

这使得这些方法是可选的，在该类的扩展中，根据数据结构的有效实现方式来对其中的一个或这三个方法进行覆盖（见上述比较表中的相关内容）。或者在扩展中就所有上述所示的实现代码，在这种情况下，异常的产生是由于应用使用了本不该使用的方法。

下面是定义在 java.util 包中的 AbstractSequentialList 类的结构。

```
public abstract class AbstractSequentialList extends AbstractionList
 {
 protected                    AbstractSequentialList ( );
 public void                  add(int index, Object  object);
 public boolean               addAll (Collection  collection);
 public boolean               addAll ( int index, Collection collection );
 public Object                get( int index );
 public Iterator              iterator();
 public int                   lastIndexOf ( Object  object);
 abstract public ListIterator listIterator( int index );
 public Object                remove( int index );
 public Object                set( int index, Object object );
}
```

方法 listIterator()和 size()均被声明为抽象方法，所以，该类的任何具体扩展类都要实现这两个方法 (抽象方法 size()是从 AbstractCollection 类继承过来的，见 5.3 节)。

8.4 线性表迭代器

迭代器是一个对象，它能从集合中的一个元素移至下一元素，它是数组或向量下标的另一选择。正如下标是处理数组的一种自然的机制，迭代器成为处理线性表的自然机制。

Java 1.2 在 java.util 包中定义了 interator 接口和 ListIterator 子接口，它们是集合框架的一部分。Interator 接口说明了一个单向迭代器在一个通用线性数据结构上的行为（见 5.5 节），而 ListIterator 子接口说明了一个在双向链接线性表上的双向迭代器的行为。

下面是 java.util 包中的 ListIterator 接口定义。

标准分享网 www.bzfxw.com 免费下载

```
public interface ListIterator extends Iterator
{ public void  add( Object object );
  public boolean hasNext( );
  public boolean hasprevious( );
  public Object  next( );
  public int     nextIndex( );
  public Object  previous( );
  public int     previousIndex( );
  public void    remove( );
  public void    set(Object object );
}
```

例8.2 线性迭代器的使用

```
import java.util.*;
public class Ex0802
{ public static void main(String[] args)
  { String[] planets = new String[]{"Venus","Earth","Mars","Pluto"};
    List list = Arrays.asList(planets);
    System.out.println("list = " + list);
    ListIterator it = list.listIterator();
    System.out.println("it.next() = " + it.next());
    System.out.println("it.next() = " + it.next());
    System.out.println("it.next() = " + it.next());
    System.out.println("it.next() = " + it.next());
    System.out.println("it.previous() = " + it.previous());
    System.out.println("it.previous() = " + it.previous());
    it.add("Saturn");
  }
}
```

该程序的运行结果是：

```
list = [Venus, Earth, Mars, Pluto]
it.next( ) = Venus
it.next( ) = Earth
it.next( ) = Mars
it.next( ) = Pluto
it.previous( ) = Pluto
it.previous( ) = Mars
Exception in thread "Main" java.lang.UnsupportedOperationException
      at java.util.AbstractList.add(AbstractList.java:153)
      at java.util.AbstractList$ListItr.add(AbstractList.java:503)
      at Ex0802.main( Ex0802.java:20 )
```

这个程序表明双向迭代器是如何工作的，它同时也表明了没有为迭代器对象 ListIterator 实现 add()方法，该对象是在 ArrayList 对象上进行循环处理的。因此，当调用 it.add("Saturn") 时造成了 UnsupportedOperationException 异常的产生。

8.5 ArrayList 类

ArrayList 类使用数组来实现 AbstractList 类。定义在 java.util 包中的 ArrayList 类如下所示：

```
public class ArrayList extends AbstractList implement List
{ public boolean   add(Object  object);
  public void      add(int index, Object  object);
  public boolean   addAll (Collection  collection);
  public boolean   addAll ( int index, Collection collection );

  public           ArrayList( );
  public           ArrayList( int  initialCapacity);
  public           ArrayList( Collection collection);
  public void      clear();
  public Object    clone( );
  public boolean   contains(Object  object);
  public void      ensureCapacity( int capacity);
  public Object    get( int index );
  public int       indexOf ( Object  object);
  public boolean   isEmpty();
  public int       lastIndexOf ( Object  object);
  public Object    remove( int index );
  public Object    set( int index, Object objec: );
  public int       size();
  public Object[ ] toArray();
  public Object[ ] toArray(Object[ ]  object);
  public void      trimToSize( );
}
```

方法 ensureCapacity(int) 和 trimToSize()专用于处理 ArrayList 对象的私有空间，整型参数是存储 ArrayList 对象元素的数组的实际大小。在该类实现的 21 个方法中，只有这 2 个和那 3 个构造函数是由该类引入的，其余 16 个均是对祖类 AbstractList、AbstractCollection 或 Object 类中的相应方法的覆盖。

例 8.3 用 ArrayList 保存等级数据

这里有一个名为 Cities.txt 的正文文件，如图 8.3 所示。该文件列出了 1990 年世界上 9 个最大的城市，并按它们的人口数的降序进行排列（如 Tokyo（东京）为 9 个最大

标准分享网 www.bzfxw.com 免费下载

城市之首)。

下面的程序将该数据文件读入一个 ArrayList 对象，并使用 indexOf()方法，remove()方法以及 add()方法来对数据进行修改，以反映 1995 年时的人口数量。

Cities.txt

```
Tokyo
Mexico City
Sao Paulo
Seoul
New York
Osaka
Bombay
Calcutta
Buenos Aires
```

图 8.3 按人口数排列的九大城市

```
import java.io.*;
import java.util.*;

public class Ex0803
{ private static ArrayList list = new ArrayList();
  public static void main(String[] args)
  { load("Cities.txt");
    System.out.println(list);
    list.remove(list.indexOf("Seoul"));
    list.remove(list.indexOf("Osaka"));
    list.add(5,"Shanghai");
    list.add(6,"Los Angeles");
    System.out.println(list);
  }
  private static void load(String filename)
  { try
    { File file = new File(filename);
      FileReader reader = new FileReader(file);
      BufferedReader in = new BufferedReader(reader);
      String city = in.readLine();
      while (city != null)
      { list.add(city);
        city = in.readLine();
      }
    }
    catch(Exception e) { System.out.println(e); }
  }
}
```

该程序的指向结果为：

```
[Tokyo, Mexico City, Sao Paulo, Seoul, New York, Osaka, Bombay,
 Calcutta, Muenos
 Aires]
[Tokyo, Mexico City, Sao Paulo, Seoul, New York, Bombay,Shanghai,Los
 Angeles, Calcutt
a, Muenos Aires]
```

8.6 LinkedList类

LinkedList 类使用链接表来实现 AbstractSequentialList 类，它位于 java.util 包中，其定义如下所示：

```
public class LinkedList extends AbstractSequentialList implements List
 {
 public boolean        add( Object object );
 public void           add(int index, Object object);
 public boolean        addAll (Collection collection);
 public boolean        addAll ( int index, Collection collection );
 public void           addFisrst( Object object);
 public void           addLast(Object  object);
 public void           clear( );
 public Object         clone( );
 public boolean        conatains( Object object );
 public Object         getFirst( );
 public Object         getLast( );
 public int            indexOf ( Object object);
 public int            lastindexOf( Object object );
 public                LinkedList( );
 publlic               LinkedList(Collection collection);
 public ListIterator   listIterator( int index);
 public Object         remove( int index );
 public boolean        remove(Object object);
 public Object         removeFirst( );
 public Object         removeLast();
 public Object         set( int index, Object object );
 public int            size( );
 public Object [ ]     toArray( );
 public Object [ ]     toArray( Object[ ] object);
}
```

在 LinkedList 类中所定义的 24 个方法中，只有 9 个加粗的方法不是对祖先方法的覆盖，其他方法均是对祖先类 AbstractSequtntialList、AbstractList、AbstarctCollection 或 Object 类中相应方法的覆盖。

例 8.4 类 Ring

这里定义一个环形双向链接线性表，它与 java.util.LinkedList 类相似，只是 next()和 previous()方法可以从线性表的一个末端绕到另一末端。

```
package schaums.dswj;
```

标准分享网 www.bzfxw.com 免费下载

```
import java.util.*;

public class Ring extends java.util.AbstractSequentialList
{ private Node header;
  private int size = 0;

  public Ring()
  {
  }

  public Ring(List list)
  { super();
    addAll(list);
  }

  public ListIterator listIterator(int index)
  { return new RingIterator(index);
  }

  public int size()
  { return size;
  }

  private static class Node
  { Object object;
    Node previous, next;

    Node(Object object, Node previous, Node next)
    { this.object = object;
      this.previous = previous;
      this.next = next;
    }
       Node(Object object)
    { this.object = object;
      this.previous = this.next = this;
    }
  }

  private class RingIterator implements ListIterator
  { private Node next, lastReturned;
    private int nextIndex;

    RingIterator(int index)
    { if (index<0 || index>size)
```

```
    throw new IndexOutOfBoundsException("Index : " + index);
  next=(size==0?null:header);
  for (nextIndex=0; nextIndex<index; nextIndex++)
    next = next.next;
}

public boolean hasNext()
{ return size > 0;
}

public boolean hasPrevious()
{ return size > 0;
}

public Object next()
{ if (size==0) throw new NoSuchElementException();
  lastReturned = next;
  next = next.next;
  nextIndex = (nextIndex==size-1?0:nextIndex+1);
  return lastReturned.object;
}

public Object previous()
{ if (size==0) throw new NoSuchElementException();
  next = lastReturned - next.previous;
  nextIndex = (nextIndex==0?size-1:nextIndex-1);
  return lastReturned.object;
}

public int nextIndex()
{ return nextIndex;
}

public int previousIndex()
{ return (nextIndex==0?size-1:nextIndex-1);
}

public void add(Object object)
{ if (size==0)
  { next = header = new Node(object);
    nextIndex = 0;
  }
  else
  { Node newNode = new Node(object,next.previous,next);
```

标准分享网 www.bzfxw.com 免费下载

```
        newNode.previous.next = next.previous = newNode;
      }
      lastReturned = null;
      ++size;
      nextIndex = (nextIndex==size-1?0:nextIndex+1);
    }

    public void remove()
    { if (lastReturned==null) throw new IllegalStateException();
      if (next==lastReturned) next = lastReturned.next;
      else nextIndex = (nextIndex==0?size-1:nextIndex-1);
      lastReturned.previous.next = lastReturned.next;
      lastReturned.next.previous = lastReturned.previous;
      lastReturned = null;
      --size;
    }

    public void set(Object object)
    { if (lastReturned==null) throw new IllegalStateException();
      lastReturned.object = object;
    }
  }
}
```

在上述程序中，Ring 类是 AbstractSequentialList 类的一个子类，它需要实现 listIterator(int)方法和 size() 方法。又因为 listIterator(int)方法必须返回一个 ListIterator 对象，因此，它必须实现 ListIterator 接口中的 9 个方法(见 8.4 节)，这个工作用一个内部类 RingIterator 类来完成。

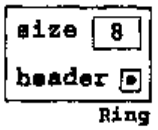


图 8.4 Ring 类示图

类 Ring 中定义了两个私有的数据字段 header 和 size，如图 8.4 所示。4 个公用方法以及两个私有类 Node 和 RingIterator。

一个链接线性表是一个链接结点序列，结点是保存线性表数据的单个对象，某个结点对应到线性表中的一个元素。因此，链接表的任何实现方式均需要一个内部结点类（Node 类），结点类应当具有一个用于保存数据的数据字段和至少一个链接字段。在本实现中，结点的数据字段是一个 Object 引用，名为 object，又因为它是一个双向链接表，结点类中有两个链接字段，名为 previous 和 next，如图 8.5 所示。在任何的链接结构中（线性表、树、图以及其他），链接字段必须是一个引用，该引用指向与结点自身具有相同类型的一个结点。在本实现中，previous 和 next 均是一个 Node 类引用。

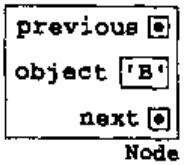


图 8.5 Node 类示图

Ring 类中的 header 字段指向线性表中的第一个结点，注意到上述实现中没有使用哑头结点（dummy ），因此结点的数目总是等于线性

表中实际的元素的个数，而元素的个数保存在该类的 size 字段中。

内部类 RingIterator 实现了 java.util.LinkedList 接口，它有 3 个字段，其中的 next 和 lastReturned 是指向线性表结点的引用，而整型数 nextIndex 用于对下标索引值的跟踪，它保持迭代器目前正锚定的结点的下标值，见图 8.6 所示。Next 字段指向线性表中的下一结点，也即如果迭代器向前移动时将访问的下一结点，而 lastReturned 字段指向最后调用 next()或 previous() 方法时所返回的结点，它定位于将由 remove()或 set ()方法调用所影响到的结点。

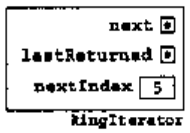


图 8.6 RingIterator 类示图

ListIterator 接口说明如果线性表在相应前进方向上还有更多的元素时，hasNext () 和 Previous()方法应当返回 true 值，对 Ring 类而言，它没有开始与结束，这是环形线性表的特性所致，只有当线性表为空时这两个方法才返回假，否则，均为真。

方法 next()和 previous()在相应方向上将迭代器移动到下一结点，语句 next = next.next; 在往前（索引值增加）的方向上使其挪动。为理解该赋值语句，应当记住一个对象的引用实际上是那个对象在内存中的地址，因此该语句将存储在 next.next 中的地址赋给引用变量 next。图 8.7 显示了例 8.5 中的测试驱动程序在运行中所生成的对象中的 12 个对象以及其实际内存地址。

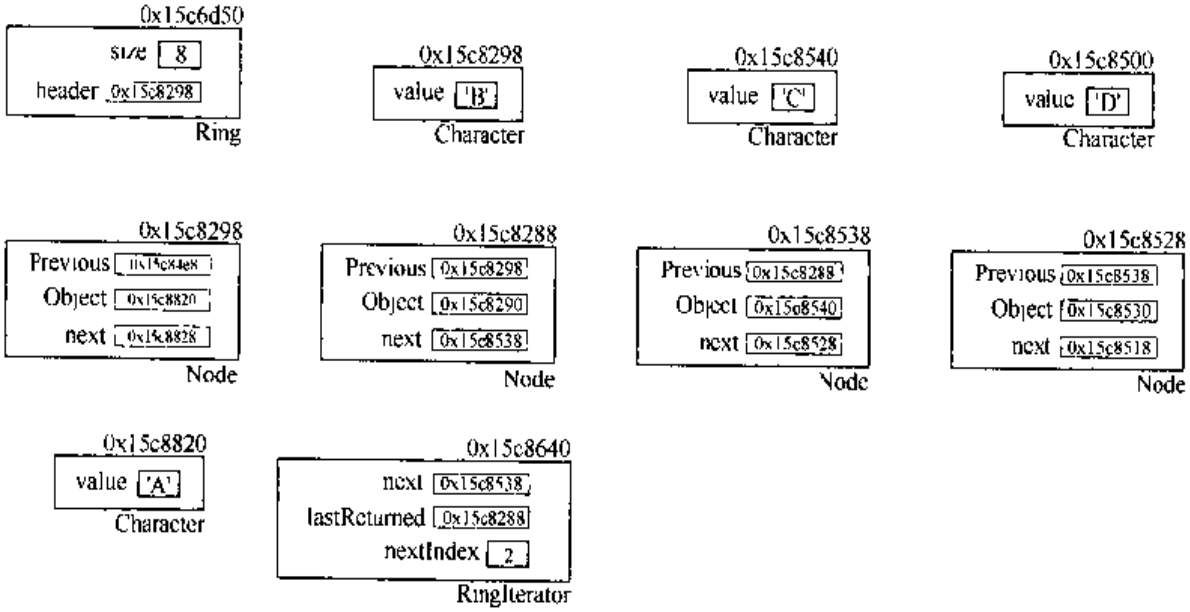


图 8.7 例 8.5 中的对象及其所在地址

如图 8.5 所示，保持在 next.next 中的地址是 0x15c8528，而保持在 next 中的地址是 0x15c8538。因此赋值语句 next = next.next; 将使 next 的值从 0x15c8538 变为 0x15c8528。换言之，它使迭代器中的 next 字段的值进行改变，从图 8.7 中所定位的第三个结点指向第四个。

注意图 8.7 中的对象在图 8.8 中的表示更为简单，每个对象引用（内存地址）的表示更为形象，用一个箭头来指向引用对象。

标准分享网 www.bzfxw.com 免费下载

另外还要注意方法 next()和 previous()中的条件表达式是如何用于 nextIndex 字段值的增加和减少的。例如，表达式：

```
nextIndex = (nextIndex == size -1 ? 0 : nextIndex + 1);
```

将根据 nextIndex 的值是否等于 size -1 而赋予不同的值，当相等时，将 nextIndex 赋值为 0，否则，使得 nextIndex 的值加 1。这是由于当 nextIndex 的值等于 size -1 时，表明迭代器到达线性表中的最后一个元素，在这种情况下，要往前移动迭代器时，将使得它绕回到线性表的第一个元素，该元素的索引值应当为 0，否则，只是简单地将其加 1 即可。

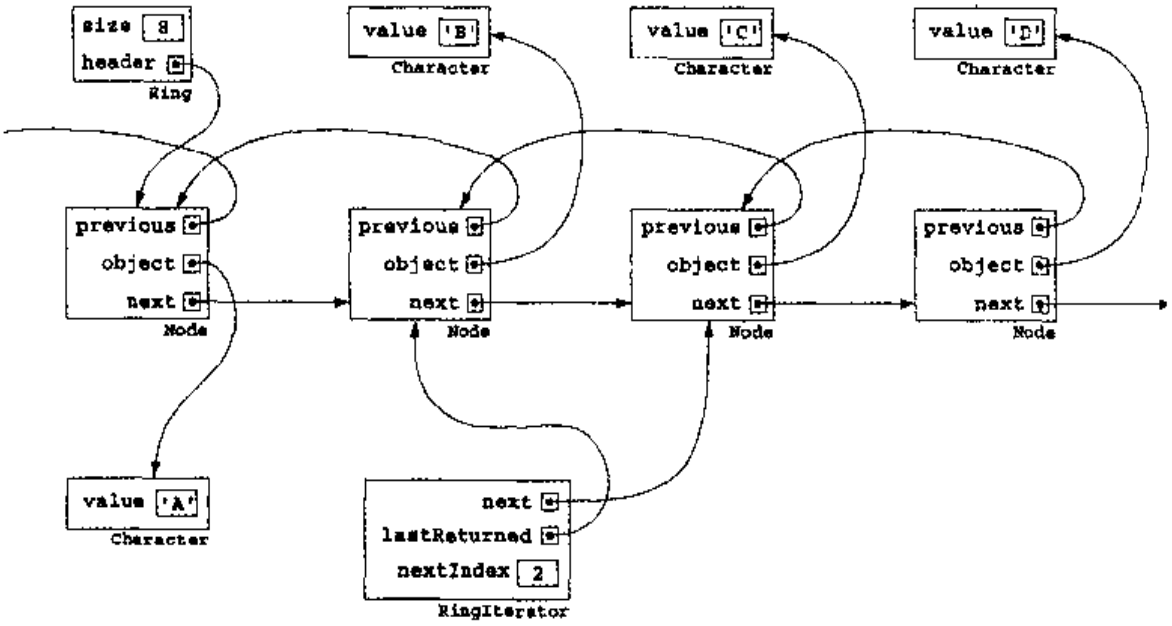


图 8.8 用指针标示的对象间的关系

方法 nextIndex() 和 previousIndex()的目的是用于分别返回下一元素和前一元素的索引值，对方法 nextIndex()而言，下一元素的索引值保存在迭代器的 nextIndex 字段中，而对 previousIndex()方法而言，在 nextIndex 字段值不为 0 的情况下，下一元素的索引值为 nextIndex - 1。而在 nextIndex 字段值为 0 的情况下，前一索引值应为 size - 1，在次反映环形链接表的绕转特性。

方法 add()将把给定对象插入到线性表中，使之成为线性表的一个新元素，下面的图说明代码

```
Node newNode = new Node(object,next.previous,next);
newNode.previous.next = next.previous = newNode;
```

是如何工作的。

要改变的指针在图 8.9 中以折线标明。注意到插入后，指针 lastReturned 是无效的（置为 null），指示迭代器执行的最后一个操作不是对 next()或 previous()的调用，这也意味着 add()或 remove()方法在下一次调用前没有再次调用 next()或 previous()方法，这样做的结果导致 UnsupportedOperationException 异常的产

生，正如例 8.2 中的情况。

例 8.5 测试 Ring 类

```
import java.util.*;
import schaums.dswj.Ring;

public class Ex0808
{ private static final int SIZE=8;
  private static Ring ring = new Ring();

  public static void main(String[]args)
  { ListIterator it = ring.listIterator();
```

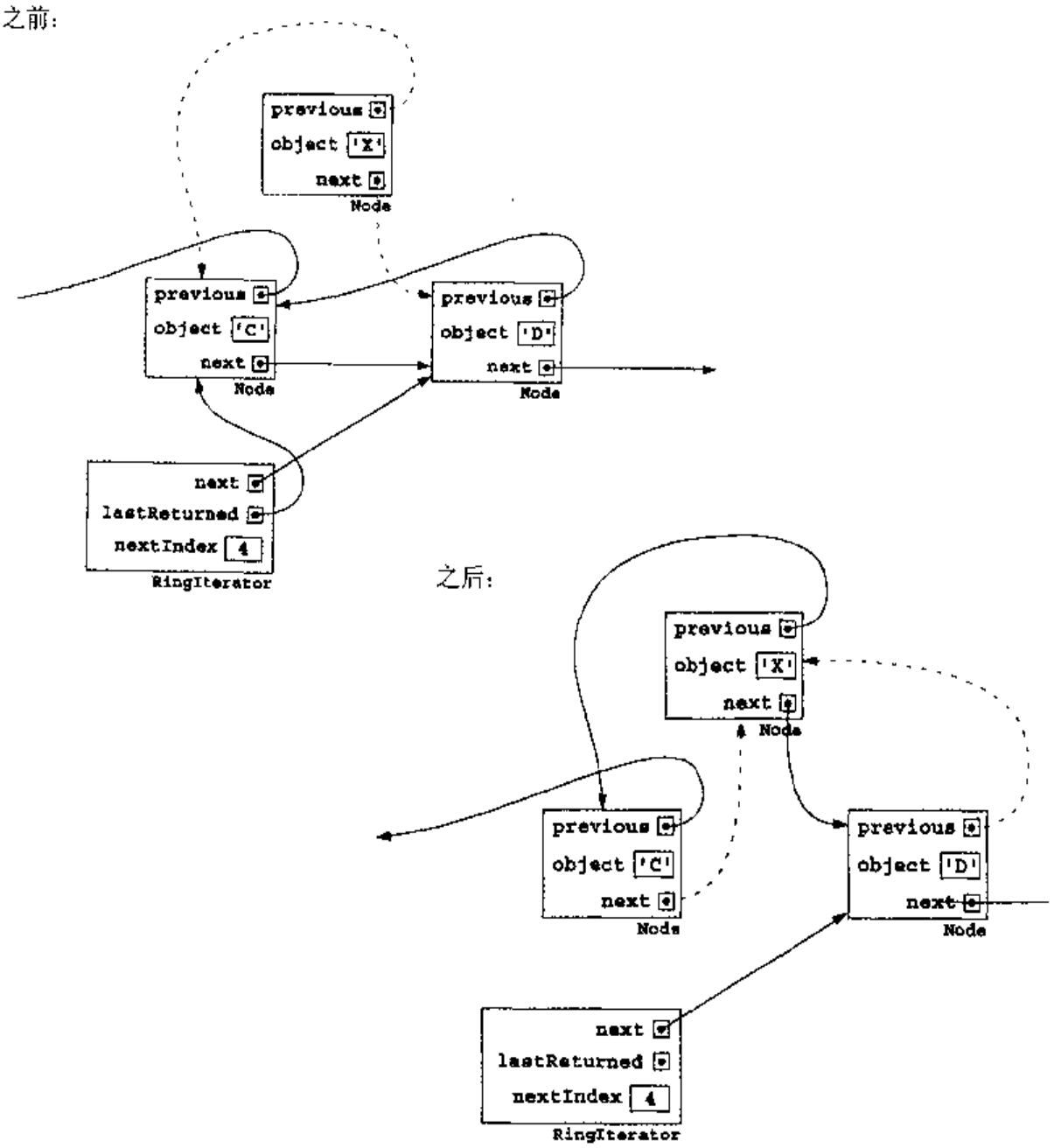


图 8.9 插入新元素时指针修改示意图

标准分享网 www.bzfxw.com 免费下载

```
    for (int n=0; n<SIZE; n++)
      it.add(new Character((char)('A'+n)));

    System.out.println(ring);
    it = ring.listIterator();  // initialize it again
    printNext(it,10);
    printPrevious(it,4);
    System.out.println(ring);
    it = ring.listIterator();  // initialize it again
    printNext(it,2);
    it.remove();    System.out.println(ring);
    printNext(it,3);
    it.remove();    System.out.println(ring);
    printPrevious(it,2);
    it.remove();    System.out.println(ring);
    printPrevious(it,3);
    it.remove();    System.out.println(ring);
  }

  private static void printNext(ListIterator it, int n)
  { for (int i=0; i<n; i++)
      System.out.println("it.nextIndex() = " + it.nextIndex()
                  + "\tit.next() = " + it.next());
  }
  private static void printPrevious(ListIterator it, int n)
  { for (int i=0; i<n; i++)
      System.out.println("it.previousIndex() = " + it.previousIndex()
                  + "\tit.previous() = " + it.previous());
  }
}
```

该程序的输出为：

```
[A, B, C, D, E, F, G, H]
it.nextIndex( ) = 0                    it.next( ) = A
it.nextIndex( ) = 1                    it.next( ) = B
it.nextIndex( ) = 2                    it.next( ) = C
it.nextIndex( ) = 3                    it.next( ) = D
it.nextIndex( ) = 4                    it.next( ) = E
it.nextIndex( ) = 5                    it.next( ) = F
it.nextIndex( ) = 6                    it.next( ) = G
it.nextIndex( ) = 7                    it.next( ) = H
it.nextIndex( ) = 0                    it.next( ) = A
it.nextIndex( ) = 1                    it.next( ) = B
it.previousIndex( ) =1                 it.previous( ) =B
it.previousIndex( ) =0                 it.previous( ) =A
it.previousIndex( ) =7                 it.previous( ) =H
```

```
it.previousIndex( ) =6                    it.previous( ) =G
[A, B, C, D, E, F, G, H]
it.nextIndex( ) = 0                       it.next( ) = A
it.nextIndex( ) = 1                       it.next( ) = B
[A, C, D, E, F, G, H]
it.nextIndex( ) = 1                       it.next( ) = C
it.nextIndex( ) = 2                       it.next( ) = D
it.nextIndex( ) = 3                       it.next( ) = E
[A, C, D, F, G, H]
it.previousIndex( ) =2                    it.previous( ) =D
it.previousIndex( ) =1                    it.previous( ) =C
[A, D, F, G, H]
it.previousIndex( ) =0                    it.previous( ) =A
it.previousIndex( ) =4                    it.previous( ) =H
it.previousIndex( ) =3                    it.previous( ) =G
[A, D, F, H]
```

为了使程序易于理解，它使用了两个打印方法printNext()和PrintPrevious()。printNext()按给定的次数，反复调用nextIndex()和next()方法。PrintPrevious()方法按给定的次数，反复调用previousIndex()和previous()方法。

首先，程序为ring对象加载8个Character对象，此时，形成的结构，如图8.10所示：

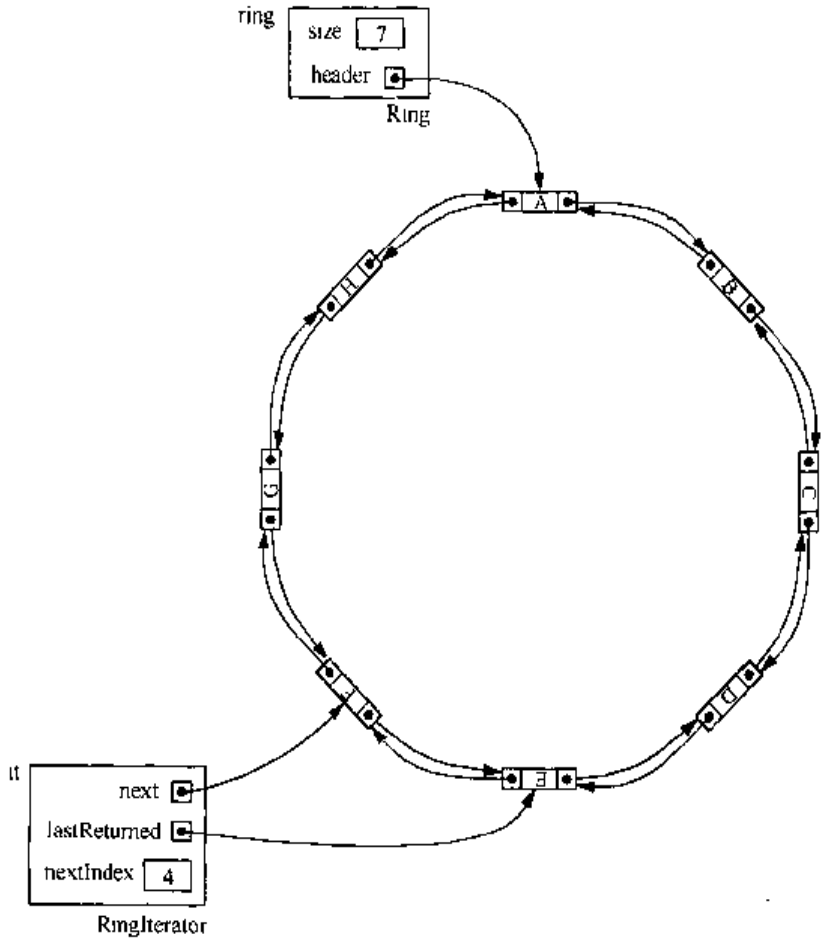


图8.10 例8.5所形成的环型线性表

标准分享网 www.bzfxw.com 免费下载

然后，它使用打印方法从顺时针和反时针来游历该环型线性表，它显示索引值是如何与结点进行匹配的。

在对迭代器进行初始化之后，它往前移动两次，因此其 lastReturned 字段指向 B 结点，然后调用 it.remove()将 B 结点移去，这正如下一个 println(ring)的输出结果中所显示的那样。类似地，再往前移动三次，然后再次调用 it.remove()，删除 E 结点。注意正如先前调用 it.remove() 一样，调用 it.remove()返回索引值 3。在第一个移出调用之前，E 结点的索引值为 4，但此时（在移出 B 结点后）仅有 3 个结点位于它的前面（A、C 和 D），所以它的索引值为 3。顺序元素的索引值总是等于位于它前面的元素的个数。

在下一个 it.remove()调用之前，进行了两次 it.previous()调用，所以，此时移出的结点的索引值为 1（C 结点）。最后，对 it.previous()的三次调用将使双向迭代器沿线性表绕转，使得 G 结点成为该移出的结点。

例 8.6 Josephus 问题

Josephus 问题建立在历史学家 Joseph ben Matthias(称为"Josephus")的一个报告的基础之上，该报告讲述了他和 40 个士兵在公元 67 年被罗马军队包围期间签订的一个自杀协定，Josephus 建议每个人杀死它的邻居，他很聪明地使自己成为这些人中的最后一个，因此获得生还，来讲述该故事。可以对该问题进行模拟，并用 Ring 类来接近该问题对 8 个士兵时的情况，见图 8.11 所示。

```
import java.util.*;
import java.io.*;
import schaums.dswj.Ring;

public class Ex0806
{ public static void main(String[] args)
  { Ring ring = new Ring();
    ListIterator it = ring.listIterator();
    int N = get("Enter number of soldiers");
    for (int k=0; k<N; k++)
      it.add(new Character((char)('A'+k)));
    System.out.print(N + " soldiers: ");
    System.out.println(ring);
    while (ring.size() > 1)
    { Object killer = it.next();
      System.out.println(killer + " killed " + it.next());
      it.remove();
    }
    System.out.println("The lone survivor is " + it.next());
  }
```

```
public static int get(String prompt)
{ int n=0;
  try
  { InputStreamReader reader = new InputStreamReader(System.in);
    BufferedReader in = new BufferedReader(reader);
    System.out.print(prompt + ": ");

    String input = in.readLine();
    n = Integer.parseInt(input);
  }
  catch(Exception e) { System.out.println(e); }
  return n;
}
```

该程序的执行结果是：

```
Enter number of Soldiers: 8
8 soldiers: [A, B, C, D, E, F, G, H]
A killed B
C killed D
E killed F
G killed H
A killed C
E killed G
A killed E
The lone survivor is A
```

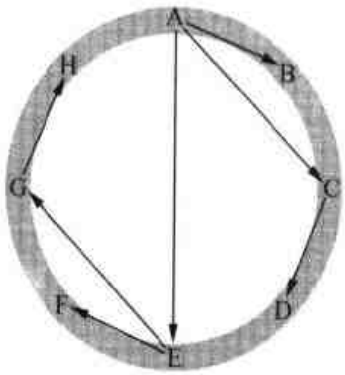


图8.11 程序运行示意图

该程序使用了以下的方法来进行整数的输入：

```
public static int get( String prompt)
{
   int n = 0;
   try
   { InputStreamReader reader = new InputStreamReader( System.in);
     BufferedReader in = new BufferedReader( reader);
     System.out.println( prompt + " : ");
     String input = in.readLine( );
     n = Integer.parseInt( input);
      }
   catch( Exception e) { System.out.println( e); }
   return n;
}
```

该方法在 schaums.dswj.IO 类中定义。

注意 Ring 只能存储对象，这一点与 Java 中的所有集合数据结构是一样的。当把整数保

标准分享网 www.bzfxw.com 免费下载

存到 ring 对象中时，我们把每个 int 值包装成 Integer 对象。

8.7 独立线性表迭代器

在同一链接线性表上，可以使用多于一个的迭代器，只要它们不对链接表的内容进行修改。

例 8.7 模拟随意行走

```
import java.util.*;
public class Ex0807
{ private static final int SIZE=26;
  private static LinkedList list = new LinkedList();
  private static Random random = new Random();

  public static void main(String[] args)
  { initializeList();
    ListIterator it1 = list.listIterator();
    ListIterator it2 = list.listIterator();
    moveForward(it1);
    moveForward(it2);
    moveBackward(it1);
    moveBackward(it2);
    moveForward(it1);
    moveForward(it2);
  }

  private static void initializeList()
  { ListIterator it = list.listIterator();
    for (int k=0; k<SIZE; k++)
      it.add(new Character((char)('A'+k)));
  }
  private static void moveForward(ListIterator it)
  { int n = random.nextInt(SIZE-it.previousIndex());
    for (int i=0; i<n; i++)
      System.out.print(it.next());
    System.out.println();
  }
  private static void moveBackward(ListIterator it)
  { int n = random.nextInt(it.nextIndex());
    for (int i=0; i<n; i++)
      System.out.print(it.previous());
    System.out.println();
```

```
    }
}
```

该程序的输出是：

```
ABCDEFGHIJKLMNOPQR
ABCDE
RQPON
ED
NOP
DEFGHIJKLMNO
```

本程序模拟在同一链接线性表上的两个独立的随意行进过程，它使用两个局部方法moveForward()和 moveBackward()来完成行进工作。对该方法的每次调用在所在的方向上进行行走的步数是随机的，它们使用了 ListIterator 迭代器中的previousIndex()和nextIndex()方法来避免跑到线性表末端之外。

在程序的本次运行中，迭代器 it1 首先向前移动 18 个元素，到达元素 R，然后迭代器 it2 独立地往前移动 5 个元素，到达元素 E。这时，迭代器 it1 往后移动 5 个元素，从 R 元素到达元素 N，而 it2 独立地往后移动 2 个元素，从 E 到达元素 D（注意当一个迭代器行进方向与原来相反时，它要对当前元素进行计数）。最后，it1 往前移动 3 个元素，从 N 到达元素 P，而 it2 则独立地往前移动 12 个元素，从 D 到达元素 O。

图 8.12 是这两个迭代器在第二次完成一系列的移动后的情景。

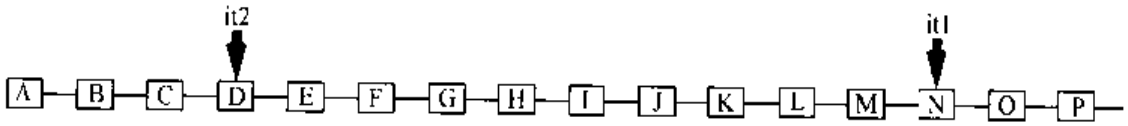


图 8.12 同一线性表上的两个独立迭代器

复 习 题

8.1 Collection 接口和 List 接口间的不同点是什么？

8.2 AbstractCollection 和 AbstractList 类之间的不同点是什么？

8.3 AbstractList 类和 AbstractSequentialList 类之间的不同点是什么？

8.4 Iterator 接口和 ListIterator 接口间的不同点是什么？

8.5 ArrayList 类和 LinkedList 类之间的不同点是什么？

8.6 一个 ArrayList 对象和 Vector 对象间的不同点是什么？

8.7 在一个应用中确定是使用 ArrayList 或是 LinkedList 时，有哪些因素可以使我们选择其中更为合适的一种？

标准分享网 www.bzfxw.com 免费下载

习 题

8.1 实现以下方法：

```
public static void loadRandomLetters( LinkedList list, int n )
// 用随机产生的 n 个大写字母来对线性表进行填充
```

8.2 用迭代器编写打印链接线性表的方法，每行只打印一个对象。

8.3 用迭代器编写一个按链接线性表逆序进行打印的方法，每行只打印一个对象。

8.4 编写以下方法：

```
public static void exchange( LinkedList list, int i, int j)
//将索引位置 i 和 j 的两元素进行交换
```

8.5 对例 8.6 中 Josephus 问题的解进行修改，使用参数 skip 来产生输出。参数 skip 的值非负整常数，它说明每个士兵应当杀谁。例如，当 skip = 2 时，A 将杀 D（跳过 B 和 C），E 将杀 H，等等。因此，原来的解则变成了一个特例，此时 skip = 0。

8.6 修改例 8.7 中的随意行走算法，使迭代器绕着 Ring 对象而不是 LinkedList（见例 8.5）对象行进，改变移到方法使得行走能绕着环对象进行。

8.7 修改例 8.7 中的随意行走算法，使迭代器绕着 Ring 对象而不是 LinkedList（见例 8.6）对象行进，并能随意改变行进方向。

复习题答案

8.1 List 接口包括以下 10 个用索引下标进行工作的方法：

```
public void          add(int index, Object object);
public boolean       addAll(int index, Collection collection);
public Object        get( int index);
public int           indexOf(Object object);
public int           lastIndexOf(Object object);
public ListIterator  listIterator( );
public ListIterator  listIterator( int index);
public Object        remove(int index);
public Object        set(int index, Object object);
public List          subList(int start, int stop);
```

8.2 AbstractList 类实现 List 接口中的方法，包括 10 个使用下标的方法，它们列在复习题 8.1 的解答中，而 AbstractCollection 类中没有这些方法。

8.3 AbstractSequentialList 类是作为链接线性表的基类而设计的，它说明两个抽象方法 listIterator()和 size()，任何具体类均应当实现这两个方法。

8.4 ListIterator 类以 AbstractSequentialList 类扩展 AbstractList 类的方式扩展 Iterator 类（见上面解答 8.2）。一般的 Iterator 对象是数组线性表上的一个无向迭代器，而 ListIterator 对象是链接线性表上的双向迭代器。

8.5 ArrayList 类的实例使用顺序、索引和直接访问（数组）等结构进行存储，而 LinkedList 类使用链接（顺序）存储结构。因此，数组线性表的访问速度更快，而链接线性表上的插入和移出等修改操作速度更快。

8.6 ArrayList 对象和 Vector 对象并没有太大的的区别，两者均提供直接索引访问。作为 Java 集合框架的一部分，ArrayList 的引入时间更晚，到 Java 1.2 才有的，因此可能更喜欢用它，而 Vector 类中的方法数大约为 ArrayList 的两倍，但许多是多余的，因而造成了一些混乱。

8.7 当需要进行经常性的查找操作时，可能更应选择 ArrayList，而在插入和移出操作经常出现的情况下，更应选择 LinkedList 对象。

习 题 解 答

8.1 实现 loadRandomLetters(LinkedList list, int n)方法：

```
public static void loadRandomLetters(LinkedList list, int n)
{  list.clear();
   while (0 < n--)
     list.add("" + (char)('A' + (int)(Math.random()*26)));
}
```

8.2 编写链接线性表输出方法：

```
public static void printForward(LinkedList list)
{ for (ListIterator itr=list.listIterator(); itr.hasNext(); )
    System.out.println(itr.next());
}
```

8.3 编写链接线性表逆序输出方法：

```
public static void printBackward(LinkedList list)
{ ListIterator itr=list.listIterator(list.size());
  while (itr.hasPrevious())
    System.out.println(itr.previous());
}
```

8.4 编写 exchange(LinkedList list, int i, int j)方法：

```
public static void exchange(LinkedList list, int i, int j)
{ Object ithObj = list.get(i);
  Object jthObj = list.get(j) ;
```

标准分享网 www.bzfxw.com 免费下载

```
    list.set(i,jthObj);
    list.set(j,ithObj);
  }
```

8.5 Josephus 问题的一般解：

```
public class Pr0805
{ public static void main(String[] args)
  { Ring ring = new Ring();
    ListIterator it = ring.listIterator();
    int n = Io.get("Enter number of soldiers");
    int skip = Io.get("Enter skip number");
    for (int k=0; k<n; k++)
      it.add(new Character((char)('A'+k)));
    System.out.print(n + " soldiers: ");
    System.out.println(ring);
    while (ring.size() > 1)
    { Object killer = it.next();
      for (int i=0; i<skip; i++)
        it.next();
      System.out.println(killer + " killed " + it.next());
      it.remove();
    }
    System.out.println("The lone survivor is " + it.next());
  }
```

8.6 修改随意行走算法，使迭代器绕着 Ring 对象行进：

```
import java.util.*;       // defines the ListIterator class
import schaums.dswj.Ring;  // defines the Ring class
public class Pr0806
{ private static final int SIZE=10;
  private static final int MAX_WALK=20;
  private static Ring ring = new Ring();
  private static Random random = new Random();

  public static void main(String[] args)
  { initializeRing();
    ListIterator it1 = ring.listIterator();
    ListIterator it2 = ring.listIterator();
    moveForward(it1);
    moveForward(it2);
    moveBackward(it1);
    moveBackward(it2);
    moveForward(it1);
    moveForward(it2);
```

```
    }
    private static void initializeRing()
    { ListIterator it = ring.listIterator();
      for (int k=0; k<SIZE; k++)
        it.add(new Character((char)('A'+k)));
    }
    private static void moveForward(ListIterator it)
    { int n = random.nextInt(MAX_WALK);
      for (int i=0; i<n; i++)
        System.out.print(it.next());
      System.out.println();
    }

    private static void moveBackward(ListIterator it)
    { int n = random.nextInt(MAX_WALK);
      for (int i=0; i<n; i++)
        System.out.print(it.previous());
      System.out.println();
    }
}
```

8.7 使迭代器以随意方向绕着Ring对象行走的程序：

```
import java.util.*;       // defines the ListIterator class
import schaums.dswj.Ring;  // defines the Ring class
public class Pr0807
{ private static final int SIZE=10;
  private static final int MAX_WALK=20;
  private static Ring ring = new Ring();
  private static Random random = new Random();

  public static void main(String[] args)
  { initializeRing();
    ListIterator it1 = ring.listIterator();
    ListIterator it2 = ring.listIterator();
    move(it1);
    move(it2);
    move(it1);
    move(it2);
    move(it1);
    move(it2);
  }

  private static void initializeRing()
```

标准分享网 www.bzfxw.com 免费下载

```
  { ListIterator it = ring.listIterator();
    for (int k=0; k<SIZE; k++)
      it.add(new Character((char)('A'+k)));
  }

  private static void move(ListIterator it)
  { int n = random.nextInt(MAX_WALK);
    boolean forward = (random.nextInt(2)==1);
    if (forward)
      for (int i=0; i<n; i++)
        System.out.print(it.next());
    else
    for (int i=0; i<n; i++)
      System.out.print(it.previous());
    System.out.println();
  }
}
```

第9章 树

树是用于层次组织结构建模的一种非线性数据结构，其基本特征是树中的每一元素可以有多个后代（也称为“孩子”），而除根外的其他元素只有惟一一个直接长辈（也称为“父亲”）。在计算机科学中，树结构是一种常用的数据结构，如计算机文件系统（见图9.1），Java类间的继承层次结构（见图9.2），Java程序在执行过程中方法调用所形成的调用关系，Java类型的分类（见图9.3），Java程序设计语言本身的语法定义等均为树型结构。

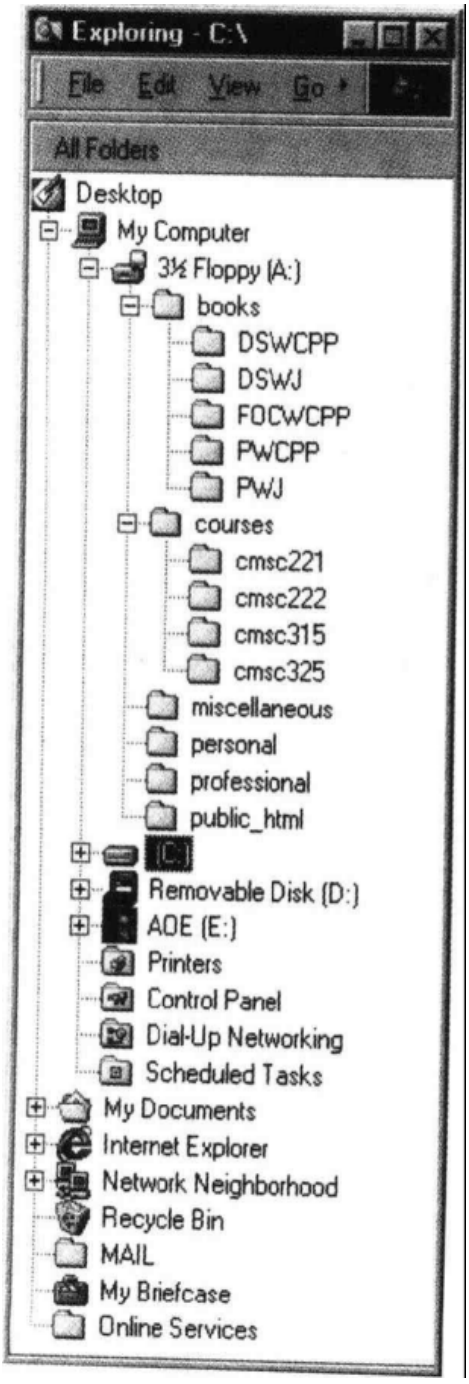


图9.1 文件系统层次结构

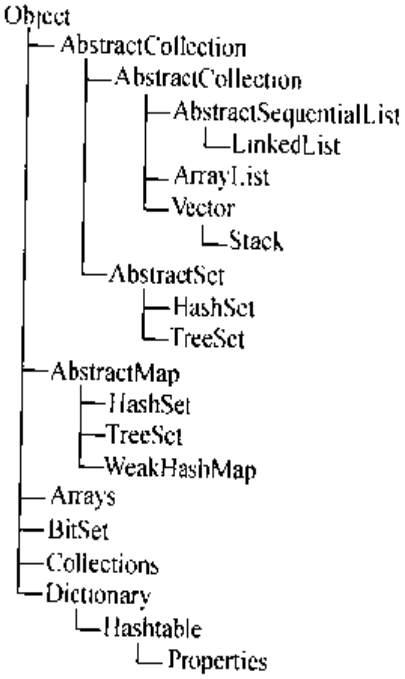


图9.2 Java类继承层次结构

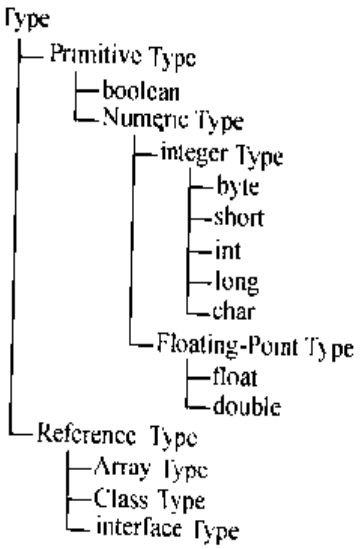


图9.3 Java类型间的层次结构

标准分享网 www.bzfxw.com 免费下载

9.1　树的定义

下面是一棵无序树的递归定义：

树是一个空集或偶对（r, S），偶对中的第一个成分 r 是一个结点，而第二个成分 S 是一个不相交的子树集合，这些子树均不包含 r 。

结点 r 称为树 T 的根，而集合 S 中的每一个元素均称为树 T 的子树。当然，集合 S 也可以是空集。对每个子树均不包含根的限制可以递归应用：r 不能在任意子树中出现，或在任意子树的任意子树中出现，等等。

注意到该定义说明树的第二个成分是一个子树集合，因此，子树间的次序是无关紧要的，这也正是称为无序树的原因。

例 9.1　相等无序树

图 9.4 所展示的两棵树是相等的。

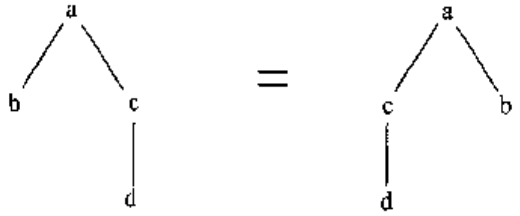


图 9.4　相等的两棵无序树

从上图可知，位于左边那棵树的根结点为 a ，并有两棵子树 B 和 C，其中 B=(b , Ø)，C = (c, { D })，子树 D 为 D = (d , Ø)。位于右边的树与左边的树的根结点是一样的，也为 a，子树集合也一样，{B, C } = { C, B}，因此，(a , { B , C }) = (a, { C , B })。

树中的元素称为树的结点。从技术上讲，每个结点只是以其为根的那棵子树的元素，间接地，树由嵌套子树组成，每个结点当成是所在嵌套树的一个元素，因此，图中的 a、b、c 和 d 均看成树 A 中的结点，同样，c 和 d 也都是树 C 的结点。

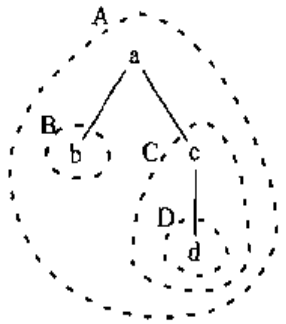


图 9.5　一棵大小为 4 的树

一棵树的大小（size）是指它所包含的结点的数目。图 9.5 所示的树 A 的大小为 4，而树 C 的大小为 2。大小为 1 的树叫作单结点树（singleton）。图 9.5 中的树 B 和 D 均是单结点树。而大小为 0 的树称为空树（empty tree），并用 Ø 表示。值得注意的是，空树是惟一的，也就是说，所有的空树均相等。

假如 $T=(r, S)$ 是一棵树，则 r 为树的根，S 为子树集合，$S=\{T_1, T_2, \ldots, T_n\}$。

每棵子树 T_j 本身又是根为 r_j 的一棵树，在这种情况下，我们称结点 r 为每个结点 r_j 的父结点，而称 r_j 为结点 r 的子结点。在一般情况下，当两结点中的一个结点是另一个的父结点

时，我们称这两结点为相邻结点（adjacent）。

没有孩子的结点为叶结点，至少有一个孩子的结点称为内部结点。

树中的一条路径（path）是一个结点序列（$x_0, x_1, x_2, .., x_m$）。其中，具有相邻下标值的每对结点（x_{i-1}, x_i）均是相邻结点。例如，(a, c, d)是上图中的一条路径，但(a, d, c)就不是一条路径。一条路径的长度为其相邻结点对的个数。

从树的定义可知，树是一种非循环结构，也就是说，树中没有包含同一结点一次以上的路径。

树中结点 x_0 的一条根路径（*root path*）是从该结点出发并最终到达根结点的一条路径，可表示为结点序列（$x_0, x_1, x_2, .., x_m$），其中 x_m 为树的根。一个叶结点的根路径称为从页到根路径（*leaf-to-root path*）。

定理 9.1 树中每一结点有惟一一条根路径

该定理的证明，见习题 9.1。

一个结点的深度（*depth*）是它的根路径的长度，当然，所有树的根结点的长度均为 0。当我们说树中子树的深度时，意指子树中的根结点的深度。

树中的层（*level*）是给定深度上所有结点的集合。

树的高度(*height*)为树中所有结点中的最大深度。本章一开始给出的 Windows 目录树结构的高度为 4。由定义可知，单结点树的高度为 0，而空树的高度为-1。

当结点 y 位于结点 x 的根路径中时，我们称结点 y 为结点 x 的祖先结点。应当注意根结点为树中所有结点的祖先结点。

但结点 y 是结点 x 的祖先结点时，我们也称结点 x 是结点 y 的一个后代结点。对树中的每个结点 y 而言，结点 y 和它的所有后代结点构成一棵以 y 为根结点的子树。如果 S 是树 T 的一棵子树，我们则称 T 是 S 的超树（*supertree*）。

树的路径长度是从根开始的所有路径的长度之和。这与树的加权值是一样的，即它等于树中的各层乘以该层中的结点数之和。在图 9.6 所示的树中，其路径长度为 1×3+2×4+3×7=32。

例 9.2 树的性质

图 9.6 中显示了一棵树的结构，它的根结点为 a, a, b, c, e, f 和 h 均是内部结点，其他 9 个结点均是叶子结点，路径（m, h, c, a）是一条从页到根路径，它的长度为 3。结点 b 的深度为 1，而结点 m 的深度为 3，第二层由结点 e, f, g 和 h 组成。

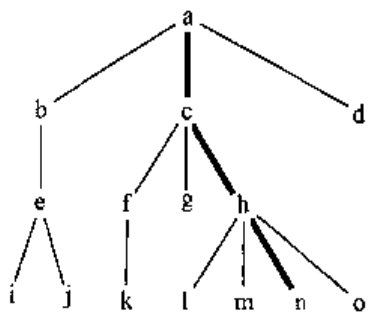


图 9.6 一棵树

树的高度为 3，结点 a, c 和 h 均是结点 m 的祖先结点，结点 k 是结点 c 的后代，但它不是结点 b 的后代。

根为 b 的子树由结点 b, e, i 和 j 组成。

一个结点的度（*degree*）是其孩子的个数。在上例中，结点 b 的度为 1，而结点 d 的度为 0，结点 h 的度为 4。

一棵树的阶（*order*）为树中所有结点的度的最大值。当一棵树的所有内部结点均有相同的

标准分享网 www.bzfxw.com 免费下载

度，并且所有的叶结点均位于同一层上时，称该树为满树（*full*）。如图 9.7 所示，树中所有结点的度均为 3，并且叶结点均位于第三层，所以，它是一棵满树。注意，该树总共有 40 个结点。

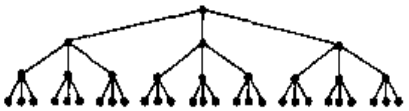


图 9.7 一棵满树

定理 9.2 一棵阶为 d，高度为 h 的满树有 $(d^{h+1}-1)/(d-1)$ 个结点

该定理的证明，见习题 9.1。

推论 9.1 一棵阶为 d 大小为 n 的满树的高度为 $h=\log_d(nd-n+1)-1$。

推论 9.2 任意一棵高度为 h 的树中的结点数最多为 $(d^{h+1}-1)/(d-1)$，d 是树中所有结点中的最大度数。

9.2 决策树与迁移图

决策树是一种树形图，该图刻画了通过一系列决策来求解一个问题的解题过程，也即该过程中所有不同的可能阶段。每个内结点用一个询问进行标注，每条弧线用对询问的一个回答来标注，每个叶结点用对问题的一个解来标注。

例 9.3 找出假硬币

有 5 个看似相同的硬币，其中一个为假币，假硬币与真硬币的不同点在于它比真硬币稍轻，惟一可用的测试是对两组硬币进行重量比较测试，为了找出该假币，应当如何选择进行重量比较的硬币组。

在如图 9.8 所示的决策树中，符号～表示两组硬币的重量比较，例如，$\{a,b\}\sim\{d,e\}$ 表示硬币 a，b 的重量和硬币 d，e 的重量进行比较。该图刻画了寻找假币的过程。

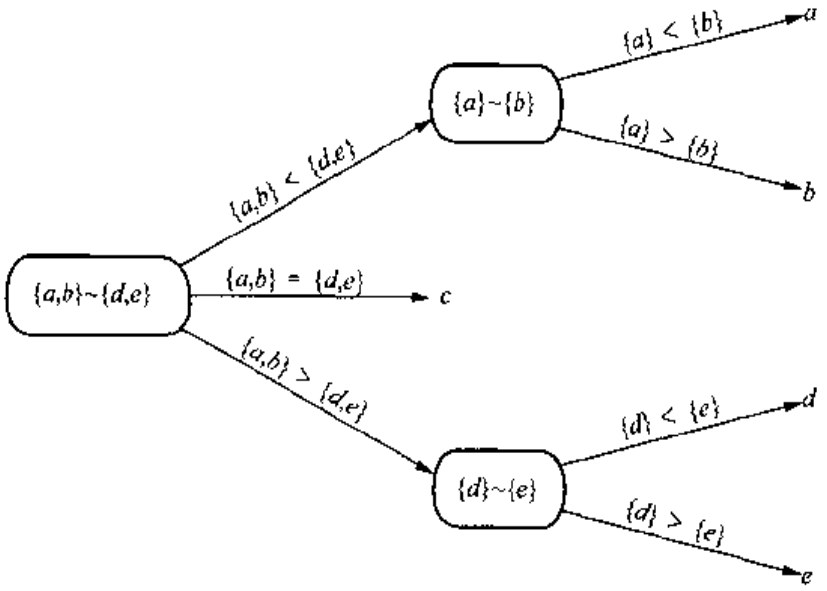


图 9.8 决策树

一个迁移图是一棵树或一个图（见第 16 章），其内结点表示一个多阶段处理过程中所处的不同的状态或局面。正如决策树，每个叶结点表示过程的一个不同输出，每个分枝用在父事件发生时该孩子事件将发生的条件概率来标注。

例 9.4 掷骰子游戏

掷骰子游戏是由两个人共同进行的，暂且称为 X 和 Y。首先 X 将两个骰子往上抛，如骰子落地后面朝上的数字之和是 7 或 11，则 X 取胜。当数字之和为 2、3 或 12 时，则 Y 取胜。否则，该数作为下一次的匹配点数。所以，当两个选手在第一次均没有取胜时，骰子重复上述丢抛过程，直到出现匹配点数，或出现数 7。当 7 先出现时，Y 取胜，否则，当匹配点数出现时，X 取胜。

图 9.9（迁移图）描述了上述掷骰子游戏过程。

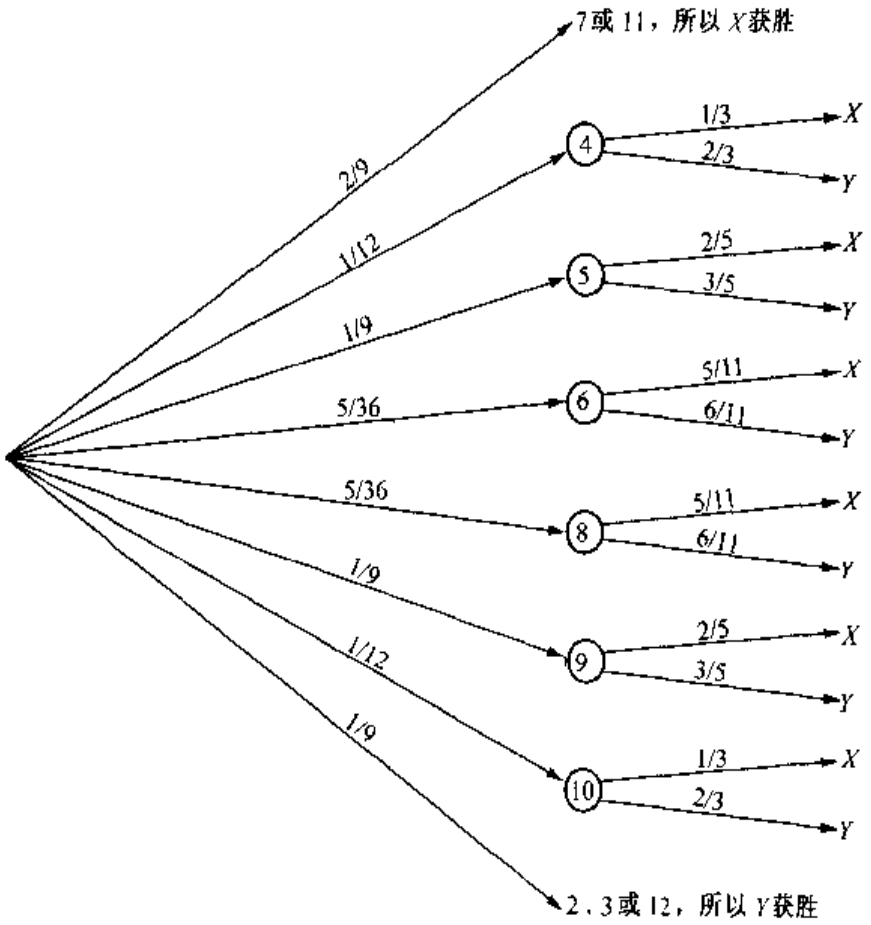


图 9.9 迁移图

当一对骰子往上抛去时，将产生 36 种不同的结果（第一个骰子有 6 种可能的数字，对每个数字而言，第二个骰子同样有 6 种可能的数字），在这 36 个结果中，数字之和为一的结果有一个（1+1），数字之和为 3 的结果有 2 个(1+2 或 2+1)，数字之和为 12 的一个（6+6）。因此，在这 36 个结果中产生 2、3 或 12 这个事件的机会为 4 次，概率为 4/36 = 1/ 9。同样，产生和数为 7 的方式有 6 种，而和数为 11 的为 2 种，因此发生事件“结果为 7 或 11”的概

标准分享网 www.bzfxw.com 免费下载

率是 8/36 = 2 / 9。在第一层上的其他概率可按同样的方式进行计算。

下面再看树中第二层上的概率值是如何通过计算获得的，我们来分析匹配点数为 4 的情况。当下一次抛出的骰子的和数为 4 时，X 取胜。当为 7 时，Y 取胜。否则，将重复上述抛骰子过程。图 9.10 对 3 种可能的情况进行了图示，概率 1/12、1/6 和 3/4 的计算过程如下：

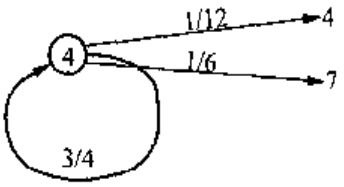


图 9.10 概率的计算

$P(4) = 3/36 = 1/12$

$P(7) = 6/36 = 1/6$

$P(2, 3, 5, 6, 8, 9, 10, 11, 12) = 27/36 = 3/4$

因此，当第一次抛出去的骰子的面上数字之和为 4 时，则在第二次抛骰子的时候，X 取胜的概率为 1/12，还在第二次分不出胜负，要进行第三次抛骰子的概率为 3/4。而 X 在第三次获胜的概率为（3/4 ）×(1/12)，而要进行第四次抛骰子的概率为（3/4）×（3/4），因此，在第一次抛出去的骰子的面上数字之和为 4 时，X 在第四次取胜的可能性为（3/4）×（3/ 4）×(1 /12)，可以依次类推，将上述部分概率求和，我们将得到在第一次抛出去的骰子的面上数字之和为 4 的条件下，X 在以后各次抛骰子的过程中取胜的概率为：

$$P_4 = \frac{1}{12} + \left(\frac{3}{4}\right)\times\frac{1}{12} + \left(\frac{3}{4}\right)^2\times\frac{1}{12} + \left(\frac{3}{4}\right)^3\times\frac{1}{12} + \left(\frac{3}{4}\right)^4\times\frac{1}{12} + \cdots$$

$$= \frac{\frac{1}{12}\times\left[1-\left(\frac{3}{4}\right)^n\right]}{1-\frac{3}{4}}, when \ \ n\to\infty$$

$$= \frac{\frac{1}{12}}{1-\frac{3}{4}}$$

$$= \frac{1/12}{1/4}$$

$$= \frac{1}{3}$$

在上述计算过程中，用到了几何级数求和公式（见定理 A.10）。

假如在第一次抛出去的骰子的面上数字之和一旦为 4 时，X 取胜的概率为 1/3。那么 Y 取胜的概率就为 2/3。这样，位于第二层上的其他概率值可按同样的方法进行计算。

现在，从上面的主迁移图 9.10 中，可以计算 X 取胜的概率值 P 了。

$$p = \frac{2}{9} + \frac{1}{12}(P_4) + \frac{1}{9}(P_5) + \frac{5}{36}(P_6) + \frac{5}{36}(P_8) + \frac{1}{9}(P_9) + \frac{1}{12}(P_{10})$$

$$= \frac{2}{9} + \frac{1}{12}\left(\frac{1}{3}\right) + \frac{1}{9}\left(\frac{2}{5}\right) + \frac{5}{36}\left(\frac{5}{11}\right) + \frac{5}{36}\left(\frac{5}{11}\right) + \frac{1}{9}\left(\frac{2}{5}\right) + \frac{1}{12}\left(\frac{1}{3}\right)$$

$$= \frac{244}{495}$$

$$= 0.4929$$

因此，*X* 取胜的可能性为 49.29%，而 *Y* 取胜的可能性为 50.71%。

一个随机过程（*stochastic process*）是这样的一个过程，该过程可以用迁移图来进行分析，也就是说，它可分解为一个事件序列，各事件的发生概率可以由计算获得。掷骰子游戏实质上是一个无穷随机过程，因为对事件的发生次数没有限制，正如例 9.4 中的分析所示，大多数随机过程可以公式化为一个等价的服从于计算机的有限随机过程。

应注意到决策树与迁移树在图示方式上与其他树不同，它通常从左到右进行绘制，这样，可以显示出从一个结点到下一结点间移动时的时间依赖关系。

9.3 有序树

下面是有序树的递归定义：

一棵有序树是一个空集或偶对（*x* , *S* ），偶对中的第一个成分 *x* 是一个结点，而第二个成分 *S* 是一个不相交的树序列，这些树均不包含 *r* 。

结点 *x* 称为树 *T* 的根，而集合 *S* 中的每一个元素均称为树 *T* 的子树。当然，集合 *S* 也可以是空集。对每个子树均不包含根的限制可以递归应用：*x* 不能在任何子树中出现，或在任意子树的任何子树中出现，等等。

注意到该定义除了包含的子树是一个序列而不是一个集合以外，其他的与无序树的定义是一致的，因此，对无序树而言，当两棵树具有相同的子树集合时，这两棵树相等。但对有序树而言，除非相同的子树集合中的子树间的次序关系也相同，否则，这两棵有序树不相等。

例 9.5 不相等的有序树

图 9.11 显示的两棵树作为有序树时是不相等的。

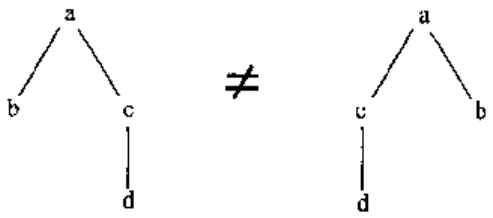


图 9.11 不相等的两棵有序树

从图 9.11 可知，位于左边那棵树的根结点为 a ，并有两棵子树 B 和 C，其中 B=(b , Ø)，C = (c, { D })，子树 D 为 D = (d , Ø)。位于右边的树与左边的树的根结点是一样的，也为 a，但是子树间的次序与左边的不一样，因为对序列而言，{B, C } ≠ { C, B}。

严格遵循该定义，可以揭示一些经常被忽视的细微之处，这正如下例 9.6 所示。

例 9.6 不相等的有序树

对 T_1= (a, (B , C))和树 T_2= (a, (B, Ø , C))这两棵树而言，它们是两棵不同的有序树，

标准分享网 www.bzfxw.com 免费下载

尽管这两棵树在图示时可以用图 9.12 表示。从技术的角度上讲，T_1 是一棵度为 2 的树，而 T_2 是一棵度为 3 的树。

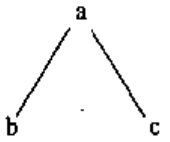


图 9.12 代表两棵不同树的同一图示

无序树的所有术语均可以以相同的方式应用到有序树上，另外，我们还可以把一棵有序树中的结点称为第一个孩子结点和最后一个孩子结点，从类比的角度把一棵有序树看成是家谱系的话往往是有帮助的，在家谱系中，孩子是按年龄排序的，首先是最为年长的，而最年轻的在最后。

9.4 有序树的树遍历算法

一个遍历算法是处理数据结构的一种算法，该算法中对该结构中的每个元素均施加一个指定的操作。例如，假如操作是打印元素的内容，则遍历算法将打印该结构中的所有元素。把操作应用到元素的过程称为访问（*visiting*）该元素，因此遍历算法的执行将使得结构中的每个元素均被访问。而元素被访问的次序取决于所使用的遍历算法，对一般树而言，有 3 个遍历它们的常用算法。

层次序遍历算法（*level order traversal*）也常称为宽度优先算法，它先访问根结点，然后访问第一层中的每一个元素，再次访问第二层中的每个元素，如此等等。每次在访问完同一层上的所有元素以后再进入下一层。如采用树中根结点在上，而叶结点位于底部的常用的图形表示方式，则层次序模式与从左到右，从上到下的阅读英文的方式是一致的。

例 9.7 层次序遍历算法

对图 9.13 中的树而言，树的层次序遍历算法将按 a，b，c，d，e，f，g，h，i，j，k，l，m 的次序访问树中各结点。

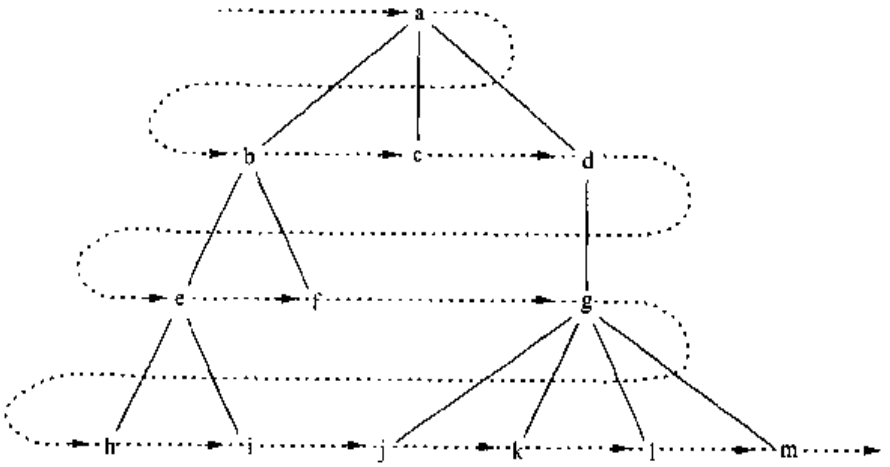


图 9.13 树

算法 9.1 有序树的层次序遍历算法

遍历一棵非空有序树的过程为：

1. 初始化一个队列。
2. 将根结点插入到队列之中。
3. 重复步骤4～7，直至队列为空。
4. 从队列中移出结点 x。
5. 访问结点 x。
6. 按顺序依次将 x 的所有孩子结点加入到队列中。

前序遍历算法（*preorder traversal*）先访问根结点，然后按子树间的次序依次以递归方式对各子树进行前序遍历。

例 9.8 前序遍历

对显示在例 9.8 中的树（见图 9.14）而言，前序遍历将按 a，b，e，h，I，f，c，d，g，j，k，l，m 的次序访问树中各结点。

注意，树的前序遍历可由对树的环绕方式来得到，从根结点开始，并对首次遇到的位于左边的每一结点进行访问处理。

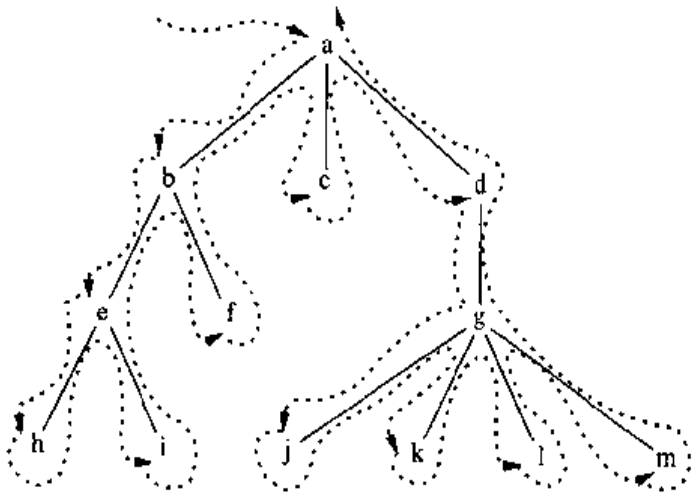


图 9.14 前序遍历示意图

算法 9.2 有序树的前序遍历

对一棵非空的有序树进行前序遍历，可按以下步骤进行：

1. 访问根结点。
2. 按子树间的次序依次以递归方式对各子树进行前序遍历。

后序遍历算法（*postorder taversal*）按子树间的次序依次以递归方式对各子树进行后序遍历，然后再对根结点进行访问。

例 9.9 后序遍历

对显示在例 9.8 中的树（见图 9.15）而言，后序遍历将按 h，I，e，f，b，c，j，k，l，m，g，d，a 的次序访问树中各结点。

标准分享网 www.bzfxw.com 免费下载

算法 9.3 一棵有序树的后序遍历

对一棵非空的有序树进行后序遍历，可按以下步骤进行：

1. 按子树间的次序依次以递归方式对各子树进行后序遍历。
2. 访问根结点。

注意层次序和前序遍历总是在访问子树中的其他结点之前先访问该子树的根结点，同时，前序遍历总是最后访问最右边的结点，而后序遍历总是首先访问最左边的结点。

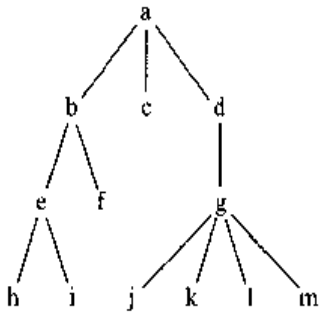


图 9.15 按后序遍历的树

前序遍历和后序遍历算法均是递归算法，它们也能通过使用一个栈结构来进行迭代实现，层次序遍历可由一个队列来加以迭代实现。

复 习 题

9.1 Java 中的所有类间的关系形成一棵树，称为 Java 继承树。

a. 在 Java1.3 中，Java 继承树的大小为多大？

b. 树的根是什么？

c. 在 Java 继承树中，什么类型的结点是 final 类？

9.2 判断下列问题的对与错。

a. 树中结点的深度等于它的祖先的个数。

b. 子树的大小等于该子树的根结点的后代数。

c. 如果 x 是 y 的后代，则 x 的深度大于 y 的深度。

d. 如果 x 的深度大于 y 的深度，则 x 是 y 的后代。

e. 一棵树是单结点树，当且仅当它的根是一个叶子结点。

f. 子树中的叶子结点也是它的超树的叶子结点。

g. 子树的根结点同样也是超树的根结点。

h. 一个结点的祖先数等于它的深度。

i. 如果 R 是 S 的子树，S 是 T 的子树，则 R 是 T 的子树。

j. 一个结点是叶子结点当且仅当它的度为 0。

k. 在任意一棵树中，内部结点的数目一定少于叶子结点的数目。

l. 一棵数是满的，当且仅当它的叶子结点位于相同的层上。

m. 满二叉树的每棵子树也是满树。

n. 满二叉树的每棵子树也是完全树。

9.3 对图 9.16 所示的 5 棵树中的每一棵树，列出它的叶子结点，结点 C 的孩子结点，结点 F 的深度，在第三层上的所有结点，树的高度和树的阶。

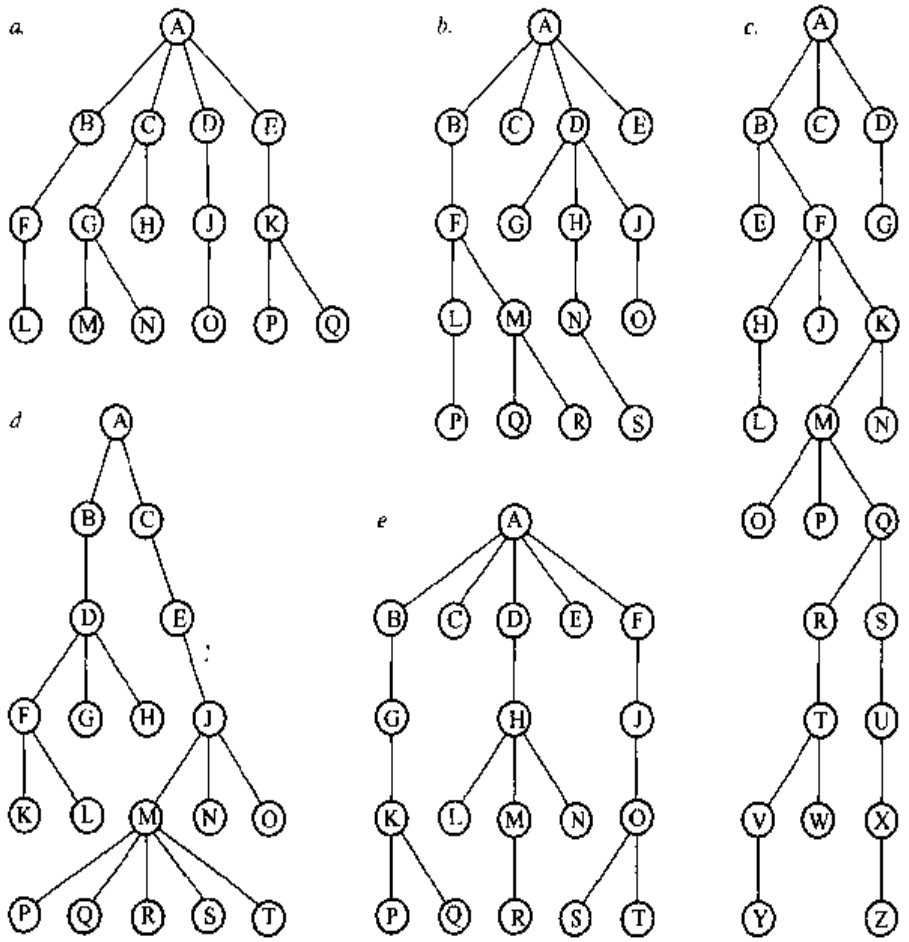


图 9.16 5 棵树

9.4 从图 9.17 中找出：

a. 结点 F 的所有祖先。

b. 结点 F 的所有后代。

c. 以根为 F 的子树中的所有结点。

d. 所有的叶子结点。

9.5 以下满树中有多少个结点：

a. 阶为 3，高度为 4？

b. 阶为 4，高度为 3？

c. 阶为 10，高度为 4？

d. 阶为 4，高度为 10？

9.6 针对例 9.2 中的树，当用以下方法进行遍历时，请分别给出树中各结点的访问次序：

a. 层次序遍历。

b. 前序遍历。

c. 后序遍历。

9.7 哪一种遍历方式总是：

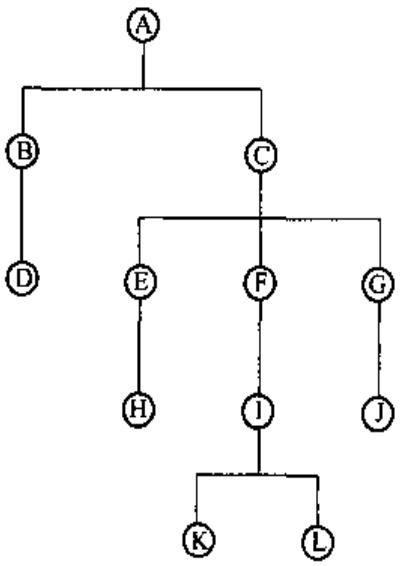


图 9.17 有祖先的树

标准分享网 www.bzfxw.com 免费下载

a. 首先访问根结点？

b. 位于最左边的结点最先访问？

c. 根最后访问？

d. 最右边的结点最后访问？

9.8 层次序遍历与读一页英文文字的模式是一致的，从左到右，从上到下逐行进行。哪一种算法与从左到右逐列阅读的模式一致？

9.9 在习题4.31的调用树中所使用的是什么遍历算法？

习 题

9.1 证明定理9.1。

9.2 证明定理9.2。

9.3 证明引理9.1。

9.4 证明引理9.2。

9.5 请推导出一棵阶为 d，高度为 h 的满树的路径长度公式。

9.6 在玩抛髓子游戏中，有些人允许 3 为一个可能点，在这种情况下，Y 在第一次抛髓子过程只有出现数字和为2或12时才算取胜。请使用迁移图来分析这种规则下的玩髓子游戏，并计算 X 取胜的概率。

9.7 圣·彼得伯格反论是一种看似能保证取胜的赌博策略，它能应用于任何双名游戏（如抛钱币时钱币的两面，人头朝上还是朝下）中，在玩的每次当中，赢与输是同等可能的，而赌的钱数是可变的。例如：在抛钱币游戏中，赌者在赌任何一次时的赌金可以是任意的美元数，当头朝上时，他将赢得所赌的钱数，反之，当尾朝上时，他将输掉所赌的钱数。圣·彼得伯格策略是下赌者继续赌一直到头出现为止，在这过程中，当没有出现头朝上时，下一次对所赌的钱数进行加倍。例如，抛出钱币头尾朝上的次序为{T, T, T, H}，则下赌者将下赌金为1美元，输，2美元，输，4美元，输，最后8美元，赢。最终净赢 -$1 -$2 -$3 -$4 +$8 = $1。因为头最终总会出现，下赌者将保证赢得1美元，而不管要进行多少次钱币的抛赌。画出该策略的迁移图，显示下赌者每次（阶段）所赢的情况，然后解释该策略的不足。

9.8 有 7 个看似相同的硬币，要对它们进行测试以确定其中的假币，区分假币与真硬的惟一特征是假币比真硬稍轻，而能使用的测试手段是用一组硬币的重量与另一组硬币的重量进行比较，怎样选择硬币组才能找出其中的假币（见例9.3）。

复习题答案

9.1 在 *Java* 继承树中：

a. 在Java1.3中，Java继承树的大小为1730。

b. 树的根是Object。

c. 在 Java 继承树中，final 类是叶子结点。

9.2 判断下列问题的对与错。

a. 对。

b. 错。它还应多一，因为子树的根在子树中，但它不是自身的后代。

c. 对。

d. 错。

e. 对。

f. 对。

g. 错。

h. 对。

i. 对。

j. 对。

k. 错。

l. 错。

m. 对。

n. 对。

9.3 *a.* 叶结点有 L，M，N，H，O，P，Q；结点 C 的孩子结点有 G 和 H；结点 F 的深度为 2；在第三层上的所有结点有 L，M，N，O，P 以及 Q；树的高度和树的阶分别为 3 和 4。

b. 叶结点有 C，E，G，O，P，Q，R 和 S；结点 C 没有孩子结点；结点 F 的深度为 2；在第三层上的所有结点有 L，M，N 和 O；树的高度和树的阶分别为 4 和 4。

c. 叶结点有 C，E，G，J，L，N，O，P，W，Y 和 Z；结点 C 没有孩子结点；结点 F 的深度为 2；在第三层上的所有结点有 H，J 以及 K；树的高度和树的阶分别为 9 和 3。

d. 叶结点有 G，H，K，L，N，O，P，Q，R，S 和 T；结点 C 的惟一孩子结点为 E；结点 F 的深度为 3；在第三层上的所有结点有 F，G，H 以及 J；树的高度和树的阶分别为 5 和 5。

e. 叶结点有 D，E，L，N，P，Q，R，S 和 T；结点 C 没有孩子；结点 F 的深度为 1；在第三层上的所有结点有 K，L，M，N 以及 O；树的高度和树的阶分别为 4 和 5。

9.4 *a.* 结点 F 的祖先为 C 和 A。

b. 结点 F 的后代为 I，K 以及 L。

c. 以根为 F 的子树中的结点为 F，I，K 和 L。

d. 叶子结点为 D，H，J，K 和 L。

9.5 *a.* 阶为 3，高度为 4 的满树中有 $(3^5-1)/2 = 121$ 个结点。

b. 阶为 4，高度为 3 的满树中有 $(4^4-1)/3 = 85$ 个结点。

c. 阶为 10，高度为 4 的满树中有 $(10^5-1)/9 = 11\,111$ 个结点。

d. 阶为 4，高度为 10 的满树中有 $(4^{11}-1)/3 = 1\,398\,101$ 个结点。

9.6 *a.* 层次序遍历时各结点的访问次序：A，B，C，D，E，F，G，H，I，J，K，L，M，

标准分享网 www.bzfxw.com 免费下载

N，O，P。

b. 前序遍历时各结点的访问次序：A，B，E，I，J，C，F，G，H，K，L，M，N，O，P，D。

c. 后序遍历时各结点的访问次序：I，J，E，B，F，G，K，L，M，N，O，P，H，C，D，A。

9.7 *a*. 层次序遍历和前序遍历总是首先访问根结点。

b. 后序遍历总是最先访问位于最左边的结点。

c. 后序遍历总是最后访问根结点。

d. 前序遍历总是最后访问最右边的结点。

9.8 中序遍历算法与从左到右逐列阅读的模式一致。

9.9 在习题4.31的调用树中所使用的是中序遍历算法。

习 题 解 答

9.1 定理9.1的证明：

假如从结点 x 到树根结点没有路径，则将违背树的定义，因为对树中的一个元素而言，x 必须是某子树的根。当有从 x 返回根的多条路径时，则 x 将成为不止一棵不同子树的元素，这也将违背树的定义，因为树的定义要求子树是不相交的。

9.2 定理9.2的证明：

当树为空时，则其高度为-1，结点数为 $n=0$。在该情况下，$n=(d^{(h)+1}-1)/(d-1)=(d^{(-1)+1}-1)/(d-1)=(d^0-1)/(d-1)=(1-1)/(d-1)=0$，因此，公式是正确的。

当树为单结点时，则其高度为 $h=0$，结点数为 $n=1$。在该情况下，$n=(d^{(h)+1}-1)/(d-1)=(d^{(0)+1}-1)/(d-1)=(d^1-1)/(d-1)=(d-1)/(d-1)=1$，因此，公式也是正确的。

现假定对任意高度为 $h-1$，$h\geqslant 0$ 的满树而言，公式是正确的。令 T 为高度为 h 的满树，由定义可知，T 由根结点和一组子树组成。又由于 T 是一棵满树，它实际由一个根结点 d 棵子树组成，每棵子树的高度为 $h-1$，因此由归纳假设可知，每棵子树的结点数为 $n_s=(d^{(h-1)+1}-1)/(d-1)=(d^h-1)/(d-1)$，所以，树 T 中的结点总数为：

$$
\begin{aligned}
n &= 1+(d)(n_s)\\
&= 1+d\left(\frac{d^h-1}{d-1}\right)\\
&= \frac{d-1}{d-1}+\frac{d^{h+1}-d}{d-1}\\
&= \frac{d^{h+1}-1}{d-1}
\end{aligned}
$$

因此，由数学归纳法原理（见A.4节）可知，该公式对任意高度的满树都是正确的。

9.3 引理9.1的证明：

该证明是纯代数演算，证明过程为：

$$n=\frac{d^{h+1}-1}{d-1}$$

$$n(d-1)=d^{h+1}-1$$

$$d^{h+1}=n(d-1)+1$$

$$=nd-n+1$$

$$h+1=\log_d(nd-n+1)$$

$$h=\log_d(nd-n+1)-1$$

9.4 引理9.2的证明。

令T为任意一棵阶为d，高度为h的树，则T可以嵌入到具有相同的阶和高度的满树中，则满树正好具有$(d^{h+1}-1)/(d-1)$个结点，因此它的子树T拥有的结点数最多为$(d^{h+1}-1)/(d-1)$个。

9.5 阶为d，高度为h的满树的路径长度是：

$$\frac{d}{(d-1)^2}\left[hd^{h+1}-(h+1)d+1\right]$$

例如，9.1节中的满树的路径长度为102。

9.6 当3是一个点的抛骰子游戏时，所形成的迁移图为图9.18。

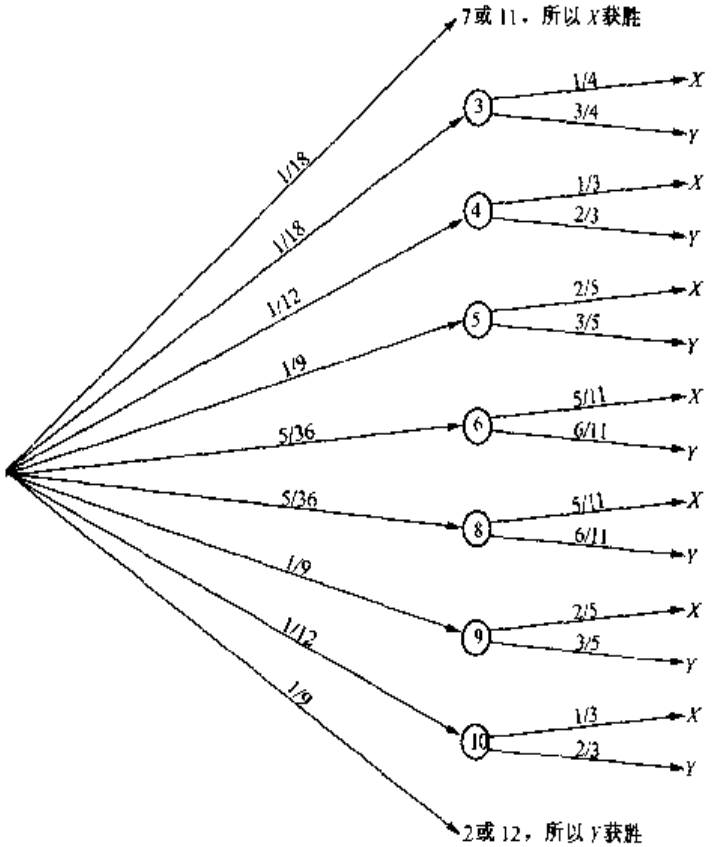


图9.18 当3为一个点值时的迁移图

标准分享网 www.bzfxw.com 免费下载

在该规则下，X 取胜的概率为 0.5068 或 50.68%。

9.7　圣·彼得伯格反论的树形图分析如图 9.19 所示：

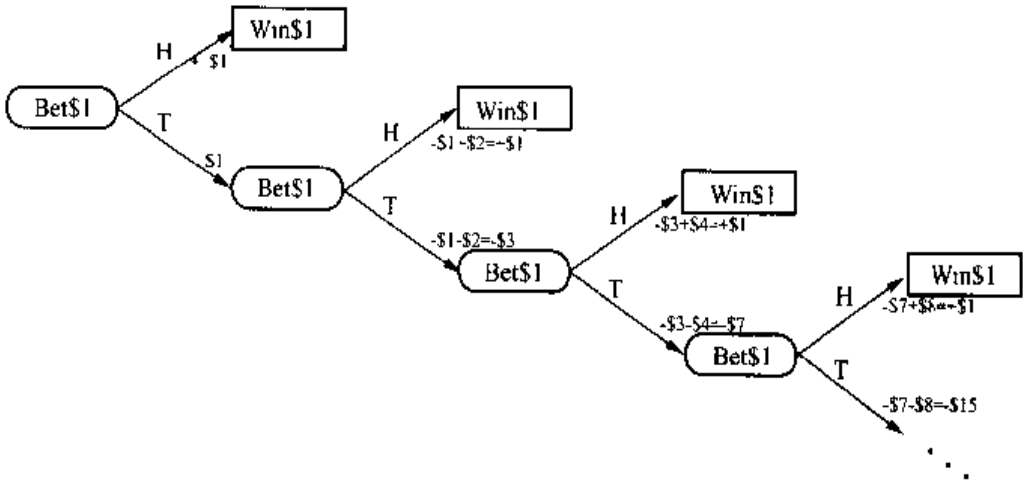


图 9.19　圣·彼得格反论树形图

该策略的不足是有一种明显的可能，即在出现头面朝上之前会出现足够多次的尾面朝上情况，使得下一次的赌金超过下赌者的赌本，这是因为在出现一系列尾面朝上的情况后，下赌者的下一次赌金为 2^n 元。例如，如果接连出现 20 次尾面朝上的情况，下一次下的赌注必须超过一百万美元。

9.8　下面的决策树 9.20 显示了解决 7 个硬币问题的所有可能的算法输出：

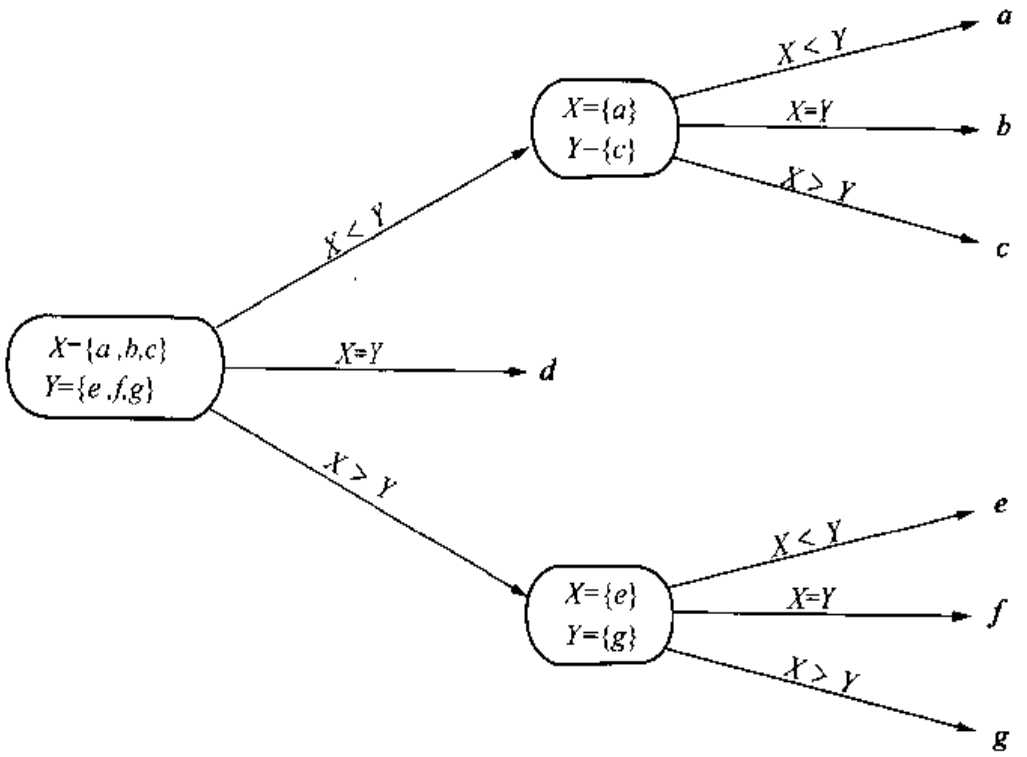


图 9.20　7 个硬币问题的决策树

第10章 二 叉 树

10.1 定义

我们首先给出二叉树的递归定义如下：

1）空集是二叉树，

2）三元组 $T=(x, L, R)$ 是二叉树，其中，x 是节点，L 和 R 是两棵均不包含 x 且不相交的二叉树。

在上面的定义中，节点 x 称为树 T 的根，子树 L 和 R 分别称为树 T 中以 x 为根的左子树和右子树。

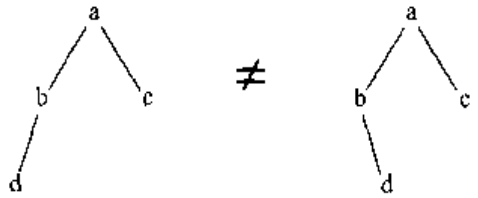


图 10.1 不同的二叉树

如果把这个定义同 9.3 节中关于排序树的定义相比较就不难发现，二叉树实际上就是一种 2 阶排序树。需要注意的是，二叉树的空左子树和空右子树是不同的（参见例 9.6）。同样，图 10.1 给出的两棵二叉树也是不同的。

下面给出同上面递归定义等价的、二叉树的非递归定义：

二叉树是所有内部节点的度都为 2 的排序树。

在这个更为简单的定义中，二叉树的叶节点的作用主要是为了维持二叉树的定义。在实际的应用中，数据一般都存储在中间节点而不是叶节点，叶节点一般实现为空节点或空引用 null。如图 10.2 所示，这种实现方案可能存在一定的效率损失或者有些复杂，但是它通常易于编程实现。

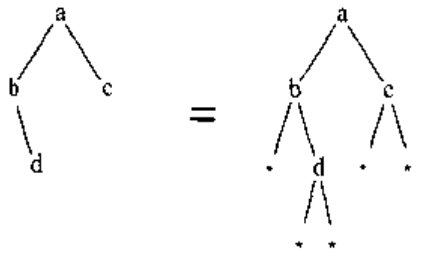


图 10.2 相同的二叉树

除非特别声明，在本书中我们将遵循第一种定义，即二叉树的递归定义。因此，有些内部节点可能只包含一棵左子树或只包含一棵右子树。

关于二叉树的一些其他术语，如大小、路径、路径长度、节点深度、层、高度、内部节点、祖先、后代、子树、超树等，同一般的树定义中的相应概念相同，这里不再赘述。

例 10.1 二叉树的特征

图 10.3 给出了一棵大小为 10、高度为 3 的二叉树。其中，节点 a 是树根，从节点 h 到节点 b 的路径长度为 2，节点 b 处于第 1 层，节点 h 处于第 3 层，b 是 h 的祖先，h 是 b 的后代，线框中的部分是一棵以节点 b

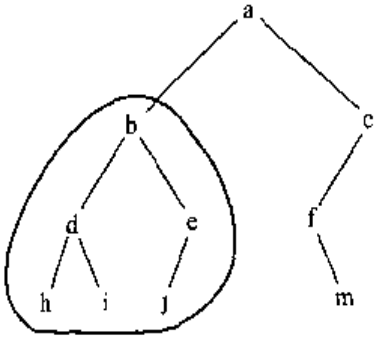


图 10.3 大小为 10、高度为 3 的二叉树

标准分享网 www.bzfxw.com 免费下载

为根的、大小为 6、高度为 2 的子树。

10.2　二叉树的计数

例 10.2　所有大小为 3 的二叉树

大小为 3 的不同的二叉树共有 5 种，如图 10.4 所示。



图 10.4　大小为 3 的二叉树

其中，4 棵树的高度为 2，1 棵树的高度为 1。

例 10.3　所有大小为 4 的二叉树

大小为 4 的不同的二叉树共有 14 种，如图 10.5 所示。

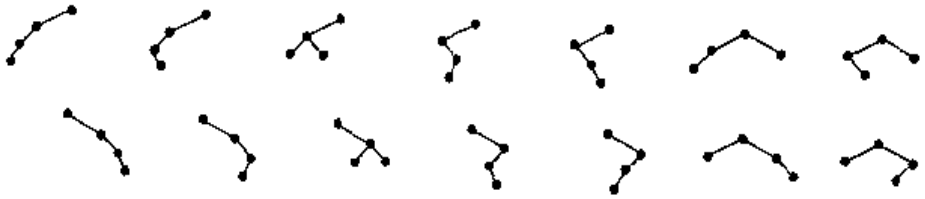


图 10.5　大小为 4 的二叉树

其中，10 棵树的高度为 3，4 棵树的高度为 2。

例 10.4　所有大小为 5 的二叉树

如果要计算大小为 5 的不同的二叉树共有多少种，可以运用二叉树的递归定义。假设 t 是一棵大小为 5 的二叉树，则它必须包括一个根节点和两棵子树且这两棵子树的大小之和为 4。那么，存在以下 5 种可能性：左子树的大小分别为 4，3，2，1 或 0 个节点。

首先，我们来计算左子树大小为 4 的情况。根据例子 10.3 可知，这样的左子树共有 14 种。此时，右子树必须为空。因此，在左子树大小为 4 的情况下，这棵大小为 5 的二叉树共有 14 种。

其次，我们再来计算左子树大小为 3 的情况。根据例子 10.2 可知，这样的左子树共有 5 种。此时，右子树必须为单个节点。因此，在左子树大小为 3 的情况下，这棵大小为 5 的二叉树共有 5 种。

再次，我们来计算左子树大小为 2 的情况。这样的左子树共有 2 种，对于每一种左子树来说，对应的右子树也应该有两种情况，因为它的大小同样是 2。因此，在左子树大小为 2 的情况下，这棵大小为 5 的二叉树共有 2×2=4 种。

依此类推，不难得出，在左子树大小为 1 的情况下，这棵大小为 5 的二叉树共有 5 种，在左子树大小为 0 的情况下，这棵大小为 5 的二叉树共有 14 种。综上，大小为 5 的不同的二叉树共有 14+5+4+5+14=42 种。

10.3 满二叉树

如果一棵二叉树满足以下性质，我们称它为满二叉树：所有叶节点都处于同层，并且所有内部节点都有两个孩子。

例 10.5 高度为 3 的满二叉树

图 10.6 给出的二叉树是一棵高度为 3 的满二叉树。它有 15 个节点，7 个内部节点，8 个叶节点。

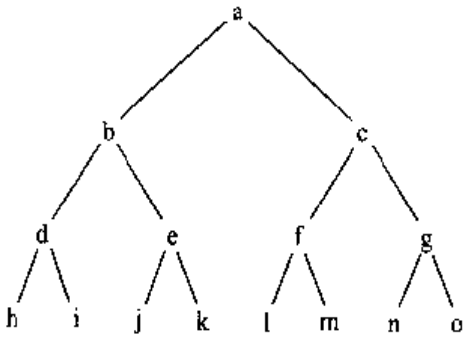


图 10.6 高度为 3 的满二叉树

定理 10.1 高度为 h 的满二叉树共有 $l=2^h$ 个叶节点和 $m=2^h-1$ 个内部节点

证明：用归纳法。首先，高度 $h=0$ 的满二叉树只有一个叶节点；因此，由于 $2^h-1=2^0-1=0$，且 $2^h=2^0=1$，命题在 $h=0$ 时成立。其次，假设 $h>0$ 且命题在树高小于 h 时都成立。考察高度为 h 的二叉树，此时其每一棵子树的高度都为 $h-1$，根据假设有 $l_L=l_R=2^{h-1}$ 及 $m_L=m_R=2^{h-1}-1$，则：

$$l = l_L + l_R = 2^{h-1} + 2^{h-1} = 2^h$$

且

$$m = m_L + m_R + 1 = (2^{h-1}-1) + (2^{h-1}-1) + 1 = 2^h - 1$$

因此，由数学归纳法原理可知，命题对于高度 $h\geq 0$ 的满二叉树都成立。

通过 m 和 h 的简单相加，可得到如下推论：

推论 10.1 高度为 h 的满二叉树共有 $n=2^{h+1}-1$ 个节点。

通过对公式 $n=2^{h+1}-1$ 作变换，容易得到如下推论：

推论 10.2 具有 n 个节点的满二叉树的高度 $h=\log_2(n+1)-1$。

我们注意到，在 $n=0$ 的情况下推论 10.2 也成立：空二叉树的高度为 $h = \log_2(n+1) - 1 = \log_2(0+1) - 1 = -1$.

标准分享网 www.bzfxw.com 免费下载

下面这个推论的导出是根据推论 10.1 以及下面的事实进行的，即在所有高度为 h 的二叉树中，满二叉树的节点最多。

推论 10.3 在任意高度为 h 的二叉树中，有：

$$h+1 \leqslant n \leqslant 2^{h+1}-1 \quad 以及 \lfloor \log_2 n \rfloor \leqslant h \leqslant n-1$$

其中，n 是二叉树的节点数。

10.4 相同、相等和同构

在计算机中，两个对象相同是指它们占据同一片内存空间，因而它们有相同的内存地址。在 Java 中相等的含义是通过等值运算符体现的。假设 x 和 y 都是对象引用，那么条件表达式(x == y)的值为 true 当且仅当 x 和 y 指向相同的对象。

在数学中，相等的概念是指两个事物具有相同的值。在 Java 中是通过 equals()方法表达的。equals()方法是在类 Object 中定义的（参见 3.4 节），能够被其他类继承，也可以在这些类中对它进行覆盖。

例 10.6 字符串的相等

```
public class Ex1006
{ static public void main(String[] args)
  { String x = new String("ABCDE");
    String y = new String("ABCDE");
    System.out.println("x = " + x);
    System.out.println("y = " + y);
    System.out.println("(x == y) = " + (x == y));
    System.out.println("x.equals(y) = " + x.equals(y));
  }
}
```

上述程序的输出为：

```
x = ABCDE
y = ABCDE
(x == y) = false
x.equals(y) = true
```

这里，对象 x 和 y（更确切地说是由变量 x 和 y 指向的两个对象）是两个不同的对象，它们在内存中占据不同的内存区域，所以对它们进行相同性测试时值为 false，即(x == y)的值为 false。但是它们具有相同的值，在数学上是相等的，所以对它们进行相等性测试的值为 true，即 x.equals(y) = true。

在 Java 中，上面所述的相同性和相等性的差异仅仅存在于引用变量（也就是说仅对对象而言）。对于其他基本数据类型的对象，等值运算符度量的都是数学概念中的相等性。

数据结构的概念既包括内容又包括结构。因此，两个数据结构可能有相同的内容但是其组织方式（即结构）不同。比如，两个数组可能都包含了 22、44 和 88 这 3 个数字，但是它们在数组中的顺序不同。

例 10.7 数组的相等

```
import java.util.Arrays;

public class Ex1007
{ static public void main(String[] args)
  { int[] x = { 44, 22, 88 };
    int[] y = { 88, 44, 22 };
    System.out.println("Arrays.equals(x,y) = " + Arrays.equals(x,y));
    Arrays.sort(x);
    Arrays.sort(y);
    System.out.println("Arrays.equals(x,y) = " + Arrays.equals(x,y));
  }
}
```

上述程序的输出为：

```
Arrays.equals(x,y) = false
Arrays.equals(x,y) = true
```

如果两个数据结构具有相同的结构，我们称它们是同构的。也就是说，它们大小相同并且包含了相同的数据，可以通过对其中数据的重组使得这两个数据结构相等。这种情况成立当且仅当在这两个结构上存在一个 · 映射。

更为准确的定义如下：

对于两个结构 X 和 Y，如果存在函数 f

1）将 X 中的每个元素 x 映射为 Y 中的惟一元素 $y=f(x)$，

2）对于 Y 中的每个元素 y 在 X 中都有惟一元素 x 与之对应，

3）X 中的元素 x_1 与 x_2 相等当且仅当 Y 中的 $f(x_1)$ 与 $f(x_2)$ 相等。

一个函数如果满足上述定义中的 3 条性质，我们称它定义了两个结构之间的同构。

需要注意的是，同构的定义并不要求两个数据结构中所包含的数据相同。同构并不对内容作要求，它只描述数据结构的结构部分。

对于数组、列表或其他一些线性数据结构，可以通过对数据结构中的数据进行某种形式的重组来使得两个同构的数据结构满足数学意义上的相等。但是对于非线性结构，如树、

标准分享网 www.bzfxw.com 免费下载

图等，这种重组就不太可能。对于两个图来说，如果它们具有相同的大小和内容但是结构不同，则无法通过对内容进行重组来让它们相等的，所以它们是非同构的。

例 10.8 同构的树

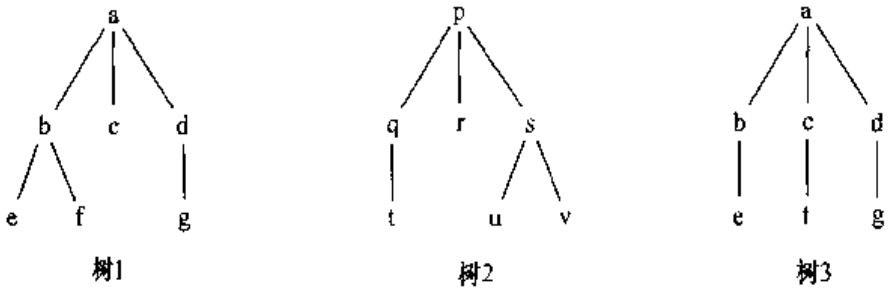


图 10.7 同构树与非同构树

如图 10.7 所示，作为非排序树，树 1 和树 2 同构但不相等。但是，树 3 与其余两棵树都不同构（当然也不相等），因为它只包含 3 个叶节点，而其余两棵树都包含 4 个叶节点。

作为排序树，树 1 和树 2 就不同构了，因为它们各自的根节点的最左子树的大小是不同的。树 1 的最左子树有 3 个节点（b、e 和 f），而树 2 的最左子树只有 2 个节点（q 和 t）。

二叉树是排序树，也就是说，对于每个节点来说，它的两个孩子的顺序也是二叉树的结构信息之一。

例 10.9 非同构的二叉树

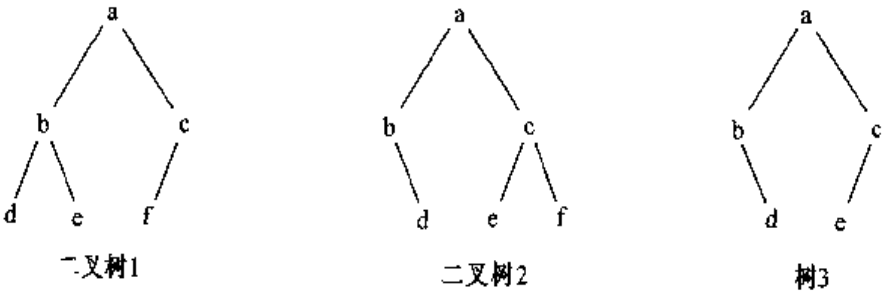


图 10.8 非同构树的二叉树

如图 10.8 所示，二叉树 1 与二叉树 2 是不同构的，原因与例 10.8 的相同：子树在顺序上是无法匹配的。但是，如果作为非排序树看待，它们却是同构的。

即使是作为非排序树，二叉树 1 和二叉树 2 都不与树 3 同构，因为树 3 包含 5 个节点，而其他两棵树都包含了 6 个节点。

10.5 完全二叉树

完全二叉树是一棵满二叉树或者是在满二叉树上仅在树的最下层的最右子树上缺少若干叶节点。

例 10.10 高度为 3 的完全二叉树

图 10.9 中的第一棵二叉树是完全二叉树，在此基础上通过在树的第 3 层增添 5 个叶节点就可以得到满二叉树，即图 10.9 中的第二棵二叉树。

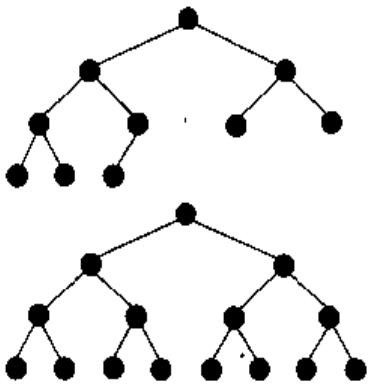


图 10.9 完全二叉树与满二叉树

定理 10.2 对于高度为 h 的完全二叉树，有

$$h+1 \leqslant n \leqslant 2^{h+1}-1 \quad \text{以及} \quad h=\lfloor \log_2 n \rfloor$$

其中，n 是二叉树的节点数。

例 10.11 完全二叉树举例

下面给出 3 个完全二叉树的例子，如图 10.10 所示：



图 10.10 完全二叉树示例

完全二叉树非常重要，因为它可以通过一般的数组进行简单、自然的编程实现。这种“自然的映射关系”实际上是定义在任意二叉树上的（参见 12.2 节）：给二叉树的根节点赋值为 1；对于其他任意节点，假设它已经被赋值为 i，则将它的左子结点（如果它存在）赋值为 $2i$，右子结点（如果它存在）赋值为 $2i+1$。按照这个方法给二叉树的每个节点赋予一个惟一的正整数值（可以看成是索引号）。这样，通过一个数组 a[]就可以方便地将任意二叉树的第 i 个节点存储在数组元素 a[i]中了。

完全二叉树可以通过上述方法进行简单的存储实现。如图 10.11 中所示，完全二叉树还可以按层存储在数组中。这种实现方法的一个非常好的地方在于，我们可以很容易地根据当前节点的索引号计算出其父节点和子节点的索引号。

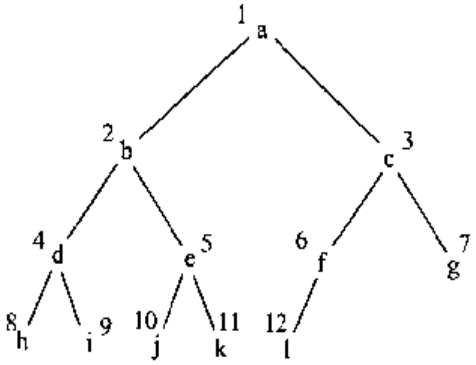


图 10.11 完全二叉树的存储映射关系

算法 10.1 完全二叉树到数组的自然映射关系

为了在以数组方式存储的完全二叉树的各节点间移动，可以按照以下步骤进行，如图 10.12 所示：

1. 存储在索引 i 处的节点的父节点存储在索引 $i/2$ 处。

0	1	2	3	4	5	6	7	8	9	10	11	12
	a	b	c	d	e	f	g	h	i	j	k	l

图 10.12 完全二叉树的存储实现示例

标准分享网 www.bzfxw.com 免费下载

2. 存储在索引 i 处的节点的左子节点存储在索引 $2i$ 处。

3. 存储在索引 i 处的节点的右子节点存储在索引 $2i+1$ 处。

例如，节点 e 存储在数组中索引号 $i=5$ 的位置上，其父节点 b 存储在索引号 $i/2=5/2=2$ 的位置上，其左子节点 j 存储在索引号 $2i=2\times5=10$ 的位置上，右子节点 k 存储在索引号 $2i+1=2\times5+1=11$ 的位置上。

实际上，完全二叉树的“完全”就是要保证这样一个条件：它在存储为数组时，数组中没有“空洞”。

例 10.12　不完全二叉树

下面给出的二叉树取自于例 10.1，这棵二叉树是不完全二叉树（如图 10.14），它在映射为数组时将在数组元素间留有“空洞”（如图 10.13 所示）。

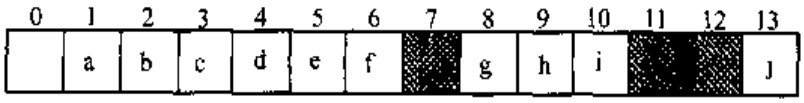


图 10.13　不完全二叉树的存储实现

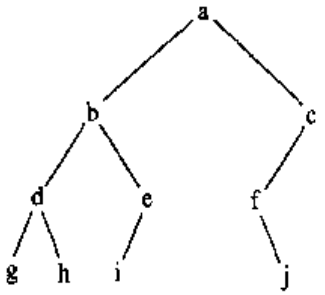


图 10.14　不完全二叉树的存储映射关系

提示：也有人用术语“近似完全二叉树”来表达本书中完全二叉树的含义；用术语“完全二叉树”来表达本书中满二叉树的含义。

10.6　二叉树遍历算法

对一般的树适用的 3 种遍历算法（参见 9.4 节）同样适用于遍历二叉树：前序遍历、后序遍历和分层遍历。此外，二叉树还允许第 4 种遍历方法：中序遍历。下面我们分别讨论这 4 种遍历算法。

算法 10.2　二叉树的分层遍历

对一棵非空二叉树进行分层遍历，可按照如下步骤进行：

1. 初始化一个队列；
2. 二叉树的根节点放入队列；
3. 重复步骤 4~7 直至队列为空；
4. 从队列头取出一个节点 x；
5. 访问节点 x；
6. 如果 x 存在左子结点，将左子结点放入队列；
7. 如果 x 存在右子结点，将右子结点放入队列。

例 10.13 二叉树的分层遍历

图 10.15 是高度为 3 的满二叉树的分层遍历示意图。

节点访问顺序依次为 A，B，C，D，E，F，G，H，I，J，K，L，M，N，O。

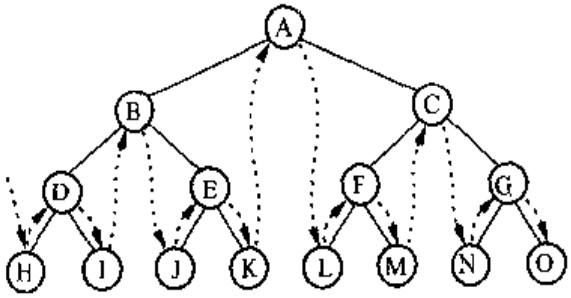


图 10.15 二叉树的分层遍历

算法 10.3 二叉树的前序遍历

对一棵非空二叉树进行前序遍历，可按照如下步骤进行：

1. 访问根节点；
2. 如果二叉树的左子树非空，对该左子树进行前序遍历；
3. 如果二叉树的右子树非空，对该右子树进行前序遍历。

例 10.14 二叉树的前序遍历

图 10.16 是高度为 3 的满二叉树的前序遍历示意图。

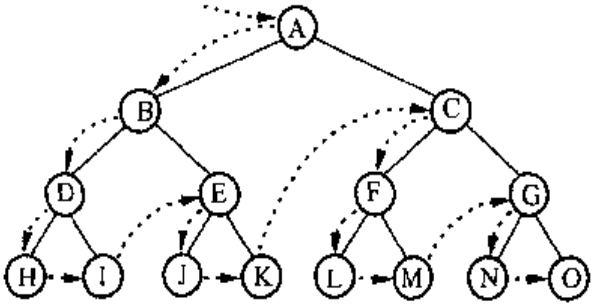


图 10.16 二叉树的前序遍历

节点访问顺序依次为 A，B，D，H，I，E，J，K，C，F，L，M，G，N，O。

请注意，二叉树的前序遍历实际上可以看成是按照一定的规则对树进行了“环绕”，这个规则就是：从根节点开始环绕过程，按照左边优先的原则，在第一次遇到其他未访问节点时，对该节点进行访问，如图 10.17 所示。

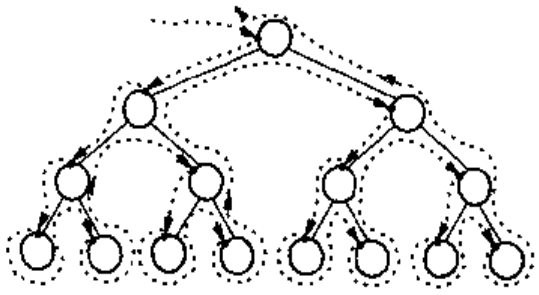


图 10.17 二叉树前序遍历节点访问顺序

算法 10.4 二叉树的后序遍历

对一棵非空二叉树进行后序遍历，可按照如下步骤进行：

1. 如果二叉树的左子树非空，对该左子树进行后序遍历；
2. 如果二叉树的右子树非空，对该右子树进行后序遍历；
3. 访问根节点；

标准分享网 www.bzfxw.com 免费下载

例 10.15 二叉树的后序遍历

图 10.18 是高度为 3 的满二叉树的后序遍历示意图。

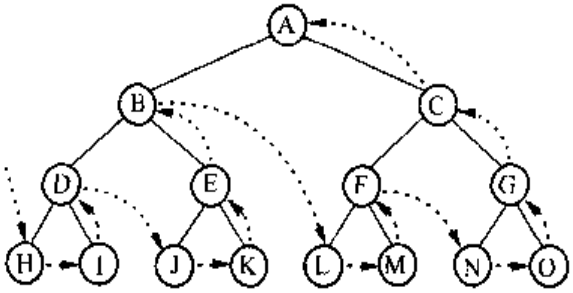


图 10.18 二叉树的后序遍历

节点访问顺序依次为 H，I，D，J，K，E，B，L，M，F，N，O，G，C，A。

对比前序遍历和后序遍历可以发现，两者的主要区别在于前序遍历首先访问根节点，而后序遍历最后访问根节点。当然，也存在第 3 种情况：在访问两棵子树之间访问根节点。这种遍历策略称为中序遍历。

算法 10.5 二叉树的中序遍历

对一棵非空二叉树进行中序遍历，可按照如下步骤进行：

1. 如果二叉树的左子树非空，对该左子树进行前序遍历；
2. 访问根节点；
3. 如果二叉树的右子树非空，对该右子树进行前序遍历；

例 10.16 二叉树的中序遍历

图 10.19 是高度为 3 的满二叉树的中序遍历示意图。

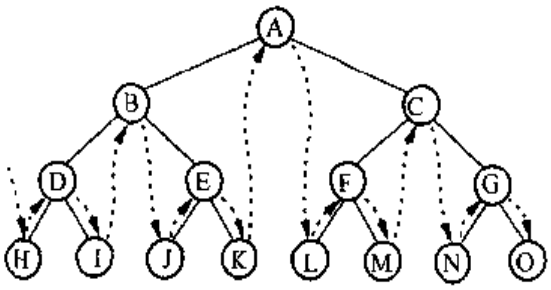


图 10.19 二叉树的中序遍历

节点访问顺序依次为 H，D，I，B，J，E，K，A，L，M，F，C，N，G，O。

10.7 表达式树

算术表达式（例如(5-x)*y+6/(x+z)）是由算术运算符（例如+，-，*，/，等等）、操作数（例如 5，x，y，6，z 等等）以及表达运算优先级的括号组成。每个算术表达式都可以由一棵二叉树惟一表示，这棵二叉树的结构是由算术运算符的运算优先级确定的。这样的二叉树称为表达式树。

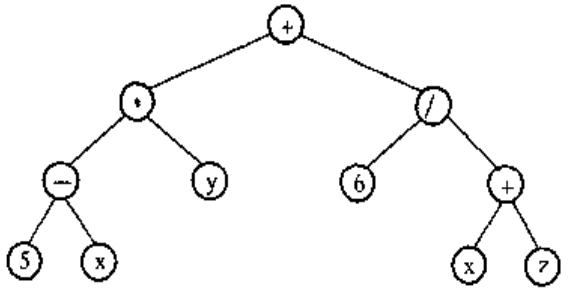


图 10.20 表达式树

例 10.17 表达式树

图 10.20 给出了表达式(5-x)*y+6/(x+z)对应的表达式树。

下面给出构造表达式树的递归算法：

算法 10.6 构造表达式树

给定表达式，其对应的表达式树可以通过如下规则递归地构造：

1. 单个算术运算符对应的表达式树由一个包含该运算符的根节点构成；

2. 假设 E_1 和 E_2 为表达式，T_1 和 T_2 分别为对应的表达式树，op 为运算符，则表达式 $E_1\ op\ E_2$ 对应的表达式树的根节点包含 op，两棵子树分别为 T_1 和 T_2。

根据遍历表达式树所用的算法的不同，表达式存在 3 种表达方式。前序遍历算法对应的是前缀表达式，中序遍历算法对应的是中缀表达式，后序遍历算法对应的是后缀表达式。后缀表达式也称为逆波兰表达式或简记为 RPN。这部分内容在 6.2 节已经讨论过。

例 10.18 表达式的 3 种表达方式

例 10.17 中的表达式的 3 种表达方式分别为：

前缀表达式：+*-5xy/6+xz

中缀表达式：5-x*y+6/x+z

后缀表达式（RPN）：5x-y*6xz+/+

一般的函数语法所使用的是前缀表达式形式。例 10.17 中的表达式如果用函数形式可以表示成：

```
sum ( product ( difference (5,x),y), quotient (6,sum(x,z)))
```

某些科学计算器使用的是 RPN，它要求在输入运算符之前要首先输入该运算符的所有操作数。

下面介绍的算法可以用来对后缀表达式进行计值。

算法 10.7 后缀表达式的计值

为了对后缀表达式进行计值，从左到右对该后缀表达式进行扫描。

1. 为运算符建立一个堆栈；
2. 重复步骤 3~9，直至扫描到表达式结束；
3. 从表达式中读取下一个元素 t；
4. 如果它是一个操作数，将它入栈；
5. 否则，转步骤 6~9；
6. 从堆栈弹出 a；
7. 从堆栈弹出 b；
8. 计算 $c = a\ t\ b$；
9. 将 c 入栈；
10. 返回堆栈的最顶元素。

标准分享网 www.bzfxw.com 免费下载

例 10.19　后缀表达式的计值

设 $x=2$，$y=3$，$z=1$，我们用上面的算法来计算例 10.18 中的表达式的值，计算过程如图 10.21 所示。

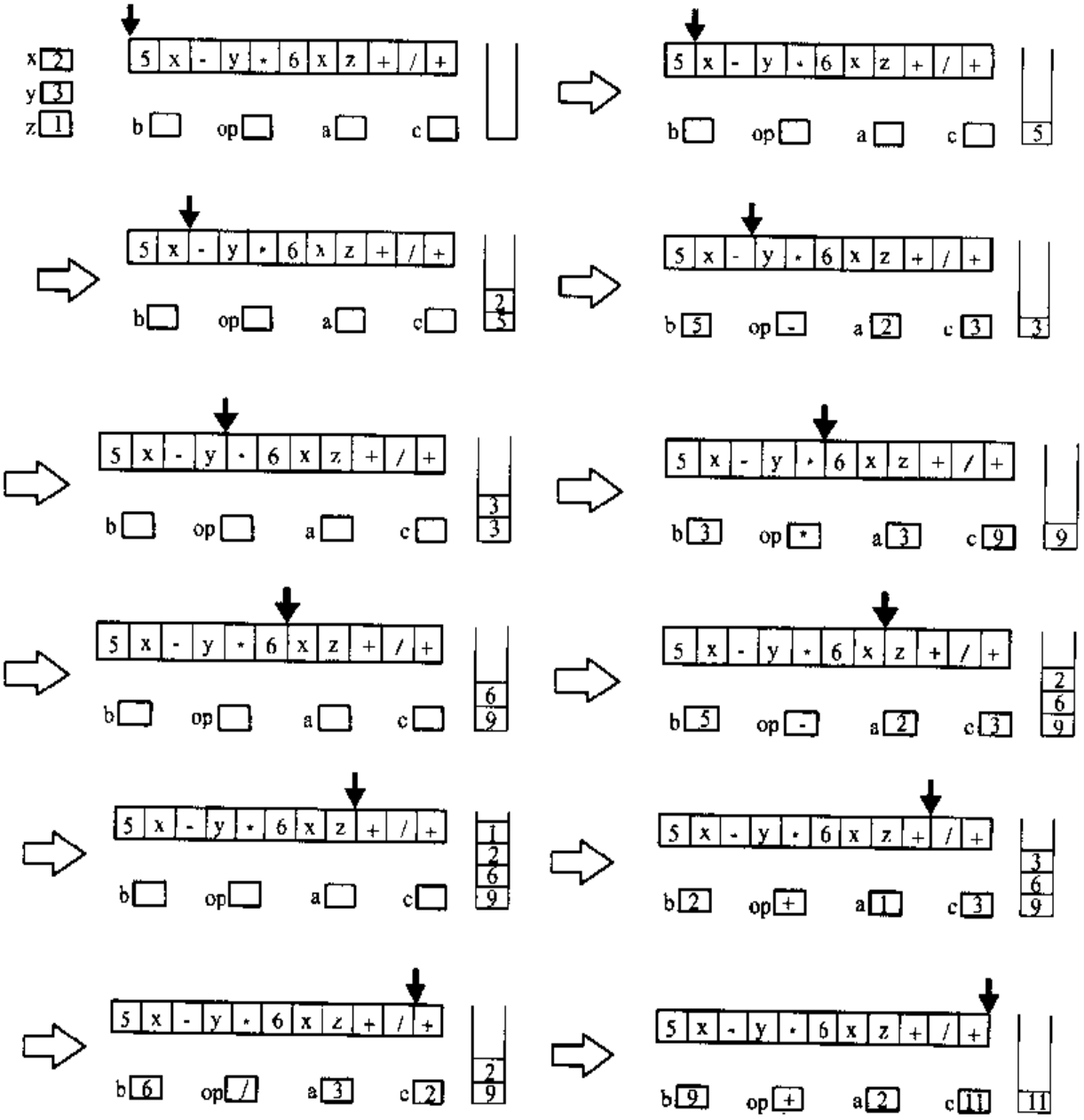


图 10.21　后缀表达式的计算

10.8　二叉树类 BinaryTree

本节给出的二叉树类 BinaryTree 是遵循二叉树的递归定义的一个具体实现。通过继承 AbstractCollection 类的实现（见 5.3），使得二叉树类 BinaryTree 与 Java Collections Framework 保持一致。

```
import java.util.*;

public class BinaryTree extends java.util.AbstractCollection
{ protected Object root;
  protected BinaryTree left, right, parent;
  protected int size;

  public BinaryTree()
  {
  }

  public BinaryTree(Object root)
  { this.root = root;
    size = 1;
  }

  public BinaryTree(Object root, BinaryTree left, BinaryTree right)
  { this(root);
    if (left != null)
    { this.left = left;
      left.parent = this;
      size += left.size();
    }
    if (right != null)
    { this.right = right;
      right.parent = this;
      size += right.size();
    }
  }

  public boolean equals(Object object)
  { if (!(object instanceof BinaryTree)) return false;
    BinaryTree tree = (BinaryTree)object;
    return ( tree.root.equals(root)
          && tree.left.equals(left)
          && tree.right.equals(right)
          && tree.parent.equals(parent)
          && tree.size == size);
  }

  public int hashCode()
  { return root.hashCode() + left.hashCode() + right.hashCode() + size;
  }
```

标准分享网 www.bzfxw.com 免费下载

```
  public java.util.Iterator iterator()
  { return new java.util.Iterator()  // anonymous inner class
    { private boolean rootDone;
      private java.util.Iterator lit, rit;  // child iterators
      public boolean hasNext()
      { return !rootDone || lit != null && lit.hasNext()
                         || rit != null && rit.hasNext();
      }
      public Object next()
      { if (rootDone)
        { if (lit != null && lit.hasNext()) return lit.next();
          if (rit != null && rit.hasNext()) return rit.next();
          return null;
        }
        if (left != null) lit = left.iterator();
        if (right != null) rit = right.iterator();
        rootDone = true;
        return root;
      }
      public void remove()
      { throw new UnsupportedOperationException();
      }
    };
  }
  public int size()
  { return size;
  }
}
```

BinaryTree 对象的构成类似于如图 10.22 所示，它包含 5 个部分：root 是 Object 对象的引用、size 是一个 int 型变量、另外还有 3 个 BinaryTree 对象的引用分别为 left、right 和 parent。二叉树的递归定义中只要求有 root、left 和 right 这 3 个变量，另外 2 个变量的设置只是为了简化部分程序。

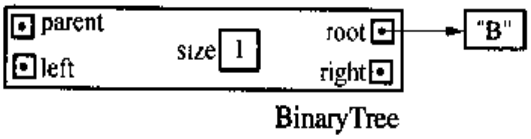


图 10.22

带 1 个参数的构造函数创建的是一个单节点树，也就是只包含根节点的二叉树。带 3 个参数的构造函数由另外两棵二叉树（都可为空）递归地创建新的二叉树。

java.util.AbstractCollection 类要求实现我们下面要分别讨论的这 4 个方法。

方法 equals (Object)覆盖了原来定义在 Object 类中的同名方法（参见 3.4 节）。同 BinaryTree 对象相等的对象 Object，必须是类 BinaryTree 的实例，而且，它的构成必须与 BinaryTree 对象相应的 5 个部分相等。

方法 hashCode()也覆盖了原来定义在 Object 类中的同名方法。它的计算逻辑很简单。

方法 iterator()覆盖了原来定义在 AbstractCollection 类中的同名（空）方法。其功能是创建一个叠代器（iterator）对象用以遍历 BinaryTree 对象。为了实现这一功能，它还在 return 语句中创建了一个私有的匿名类 Iterator。

```
return new java.util.Iterator()  // anonymous inner class
   { private boolean rootDone;
     private java.util.Iterator lit, rit;  // child iterators
     public boolean hasNext()…
     public Object next()…
     public void remove()…
   };
```

这个匿名类体包含在紧随构造函数 Iterator()之后的花括号中。值得注意的是，这段代码后面必须跟分隔符“;”，因为它标志着 return 语句的结束。整个这个构造函数看似一个方法定义，其实不然，它是一个完整的类的定义，只不过被嵌入在 return 语句中罢了。

为了返回 Iterator 对象，该匿名类必须实现 Iterator 接口（参见 5.5 节），即必须定义以下 3 个方法：

```
public boolean hasNext( ) ...
public Object next( ) ...
public void remove( ) ...
```

hasNext()方法和 next()方法都是以递归方式实现的。hasNext()方法在其方法体中激活两棵子树叠代器对象（即 lit 和 rit）的 hasNext()方法，next()方法同样也在其方法体中激活两个子树叠代器对象(即 lit 和 rit)的 next()方法。局部变量 rootDone 是用来记录根节点是否已经被叠代器访问的标识。

hasNext()方法在树的 3 部分（即树根、左子树和右子树）都被访问后才返回 false，否则都返回 true。前面已经提及，这是通过递归地使用叠代器对象 lit 和 rit 实现的。

next()方法也递归使用了 lit 和 rit 两个叠代器对象。如果根节点已被访问，则叠代器访问左子树中的下一节点（如果它存在的话），接着访问右子树中的下一节点（如果它存在的话）。如果根节点未被访问，则说明当前这棵子树的叠代器对象是第一次被调用，则要做的工作就是初始化叠代器 lit 和 rit，设置 rootDone 标识的值，最后返回根节点。

remove()方法没有实现，因为从二叉树中删除一个中间节点并不是一件简单的事情，涉及到某些策略。

例 10.20 测试 BinaryTree 类

下面给出测试 BinaryTree 类的测试程序：

```
import java.util.*;
import schaums.dswj.BinaryTree;
public class Ex1019
{ static public void main(String[] args)
```

标准分享网 www.bzfxw.com 免费下载

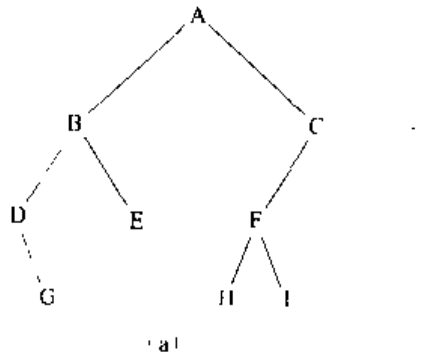
```
{ BinaryTree e = new BinaryTree("E");

    BinaryTree g = new BinaryTree("G");
    BinaryTree h = new BinaryTree("H");
    BinaryTree i = new BinaryTree("I");
    BinaryTree d = new BinaryTree("D",null,g);
    BinaryTree f = new BinaryTree("F",h,i);
    BinaryTree b = new BinaryTree("B",d,e);
    BinaryTree c = new BinaryTree("C",f,null);
    BinaryTree tree = new BinaryTree("A",b,c);
    System.out.println("tree = " + tree);
  }
}
```

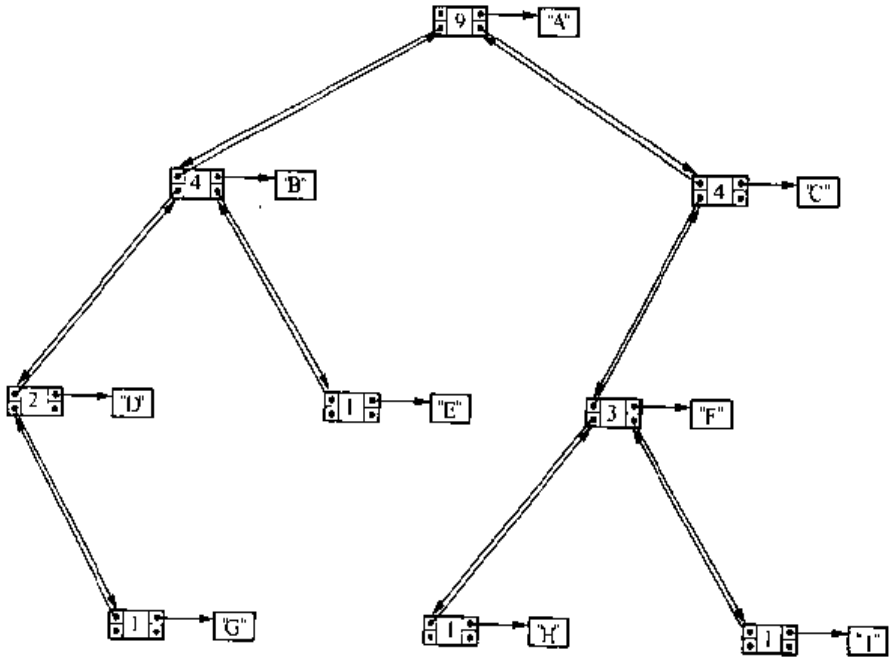
上述程序的输出为：

```
tree = [A, B, D, G, E, C, F, H, I]
```

上面这段测试程序首先创建图 10.23 所示的一棵二叉树，然后激活了 toString()方法（这一方法是从 AbstractCollection 类中继承而来的）。



(a)



(b)

图 10.23　二叉树类 BinaryTree 示例

图 10.23 从两个视角描绘了同一棵二叉树。图 10.23（b）描绘了二叉树的详细结构，其中，对象引用表示为带箭头的连线。

通过继承 AbstractCollection 类，BinaryTree 类自动继承了如下方法，它们都是在使用方法 iterator()和 size()时被定义的。

```
public boolean                  isEmpty()
public boolean                  contains(Object object)
public Object[]                 toArray()
public Object[]                 toArray(Object [] objects)
public String                   toString()
public boolean                  add(Object object)
public boolean                  addAll(Collection collection)
public void                     clear()
public boolean                  containsAll(Collection collection)
public boolean                  remove(Object object)
public boolean                  removeAll(Collection collection)
public bollean                  retainAll(Collection collection)
```

其中，涉及到对二叉树的结构进行修改的方法应该抛出 UnsupportedOperationException 异常，因为它们调用了一些未实现的方法，如 add()方法或 Iterator.remove()方法。

例 10.21 测试二叉树的 contains 方法

这个例子创建了一棵与例 10.20 相同的二叉树，然后分别调用了这棵二叉树及其子树上的 contains()方法：

```
import java.util.*;
import schaums.dswj.BinaryTree;

public class Ex1020
{ static public void main(String[] args)
  { BinaryTree e = new BinaryTree("E");
    BinaryTree g = new BinaryTree("G");
    BinaryTree h = new BinaryTree("H");
    BinaryTree i = new BinaryTree("I");
    BinaryTree d = new BinaryTree("D",null,g);
    BinaryTree f = new BinaryTree("F",h,i);
    BinaryTree b = new BinaryTree("B",d,e);
    BinaryTree c = new BinaryTree("C",f,null);
    BinaryTree a = new BinaryTree("A",b,c);
    System.out.println("a = " + a);
    System.out.println("a.contains(\"H\") = " + a.contains("H"));
    System.out.println("b = " + b);
    System.out.println("b.contains(\"H\") = " + b.contains("H"));
    System.out.println("c = " + c);
    System.out.println("c.contains(\"H\") = " + c.contains("H"));
```

标准分享网 www.bzfxw.com 免费下载

```
  }
}
```

上述程序输出为：

```
a = [A, B, D, G, E, C, F, H, I]
a.contains("H") = true
b = [B, D, G, E]
b.contains("H") = false
c = [C, F, H, I]
c.contains("H") = true
```

子树 b 和 c 在图 10.24 中标出。

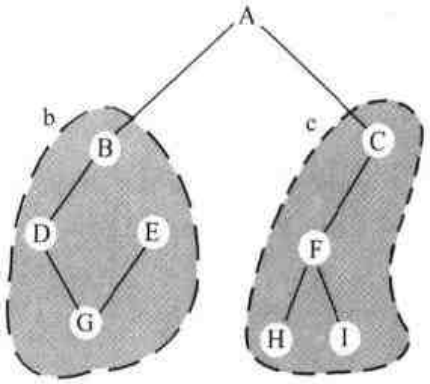


图 10.24 创建二叉树

10.9 遍历算法的实现

由 iterator()方法返回的叠代器对象在遍历二叉树时使用的是前序遍历算法（参见10.5 节）。下面对 BinaryTree 的修改将实现其他遍历算法。

```
Public class BinaryTree extends java. util.Abstract Collection
{  private Object root;
   private BinaryTree left,right,parent;
   private int size;
   //Include the same code here as is in Section 10.8 on page 192

      abstract public class Iterator
      {  protected boolean rootDone;
         protected Iterator lit,rit;// child iterators
         public boolean hasNext()
      {  return !rootDone | | lit !=null && lit.hasNext()
                  | | rit !=null && rit.hasNext();
      }
      abstract public Object next();
      public void remove()
      {  throw new UnsupportedOperationException();
      }
}
public class preOrder extends Iterator
{  public preOrder()
   {  if (left !=null) lit=left.new preOrder();
      if(right !=null)rit=right.new preOrder();
   }
  public Object next()
  {  if (! rootDone)
     {  rootDone = true;
        return root;
     }
```

```
    if (lit ! = null && lit.hasNext()) return lit.next();
    if (rit ! = null && rit.hasNext()) return rit.next();
    return null;
  }

}
public class InOrder extends Iterator
{ public InOrder()
  { if (left != null) lit = left.new InOrder();
    if (right != null) rit = right.new InOrder();
  }
  public Object next()
  { if (lit != null && lit.hasNext()) return lit.next();
    if (!rootDone)
    { rootDone = true;
      return root;
    }
    if (rit != null && rit.hasNext()) return rit.next();
    return null;
  }
}
public class PostOrder extends Iterator
{ public PostOrder()
  { if (left != null) lit = left.new PostOrder();
    if (right != null) rit = right.new PostOrder();
  }
  public Object next()
  { if (lit != null && lit.hasNext()) return lit.next();
    if (rit != null && rit.hasNext()) return rit.next();
    if (!rootDone)
    { rootDone = true;
      return root;
    }
    return null;
  }
}
public class LevelOrder extends Iterator
{ ArrayQueue queue = new ArrayQueue();
  public boolean hasNext()
  { return (!rootDone | | !queue.isEmpty());
  }
  public Object next()
  { if (!rootDone)
    { if (left != null) queue.enqueue (left);
```

标准分享网 www.bzfxw.com 免费下载

```
        if (right != null) queue.enqueue (right);
        rootDone = true;
        return root;
      }
      if (!queue.isEmpty())
      {  BinaryTree tree = (BinaryTree)queue.dequeue();
        if (tree.left != null) queue.enqueue (tree.left);
        if (tree.right != null) queue.enqueue (tree.right);

        return tree.root;
      }
      return null;
    }
  }
  }
```

首先，我们定义了一个内部的抽象类 Iterator，用来作为其他 4 个具体叠代器类的基类。它也声明了 3 个成员（即 rootDone、rit 和 lit）。需要注意的是，在 BinaryTree 外实际上并不会存在这个抽象的 Interator 类与 java.util.Iterator 类的名字冲突，因为这个抽象类是在 BinaryTree 类中定义的，因此，它的引用方式应该是BinaryTree.Iterator(参见例10.22)。在BinaryTree类的内部，同样不会存在名字冲突，因为我们使用了 java.util 包名作为 java.util.Iterator类的前缀。

抽象类 Iterator 中的 hasNext()和 remove()方法的定义同前面的匿名类版本一样。next()方法被定义为抽象的，因为4种遍历算法有不同的实现。

类 PreOrder 在其构造函数中将叠代器 lit 和 rit 定义为 PreOrder 类型的，以确保二叉树的递归遍历采用前序遍历算法。前序遍历算法（参见 10.3 节）首先访问根节点，然后依次分别对其左、右子树递归调用前序遍历算法。下面程序段中的 if 语句完成这一功能：

```
if (!rootDone)
{  rootDone = true;
   return root;
}
if (lit != null && lit.hasNext()) return lit.next();
if (rit != null && rit.hasNext()) return rit.next();
```

类 PreOrder、InOrder 和 PostOrder 的惟一区别就在于它们各自的构造函数中对递归叠代器 lit 和 rit 的定义以及 next()方法中的那 3 条 if 语句的次序不同。中序遍历对根节点的访问是在两次递归遍历之间，后序遍历对根节点的访问是在两次递归遍历之后进行的。（“Pre”表示“前”，“in”表示“中”，“Post”表示“后”）

类 LevelOrder 同其他 3 个类都不同，它没有用递归实现，而是使用了队列机制。

例 10.22 测试遍历算法

```
import java.util.*;
import schaums.dswj.BinaryTree;

public class Ex1022
{ static public void main(String[] args)
  { BinaryTree e = new BinaryTree("E");
    BinaryTree g = new BinaryTree("G");
    BinaryTree h = new BinaryTree("H");
    BinaryTree i = new BinaryTree("I");
    BinaryTree d = new BinaryTree("D",null,g);

    BinaryTree f = new BinaryTree("F",h,i);
    BinaryTree b = new BinaryTree("B",d,e);
    BinaryTree c = new BinaryTree("C",f,null);
    BinaryTree tree = new BinaryTree("A",b,c);
    System.out.println("tree = " + tree);
    BinaryTree.Iterator it;
    System.out.print("PreOrder Traversal:   ");
    for (it = tree.new PreOrder(); it.hasNext(); )
      System.out.print(it.next() + " ");
    System.out.print("\nInOrder Traversal:    ");
    for (it = tree.new InOrder(); it.hasNext(); )
      System.out.print(it.next() + " ");
    System.out.print("\nPostOrder Traversal:  ");
    for (it = tree.new PostOrder(); it.hasNext(); )
      System.out.print(it.next() + " ");
    System.out.print("\nLevelOrder Traversal: ");
    for (it = tree.new LevelOrder(); it.hasNext(); )
      System.out.print(it.next() + " ");
    System.out.println();
  }
}
```

上述程序输出为：

```
tree = [A, B, D, G, E, C, F, H, I]
PreOrder Traversal:   A B D G E C F H I
InOrder Traversal:    D G B E A H F I C
PostOrder Traversal:  G D E B H I F C A
LevelOrder Traversal: A B C D E F G H I
```

10.10 森林

树的列表构成森林。

标准分享网 www.bzfxw.com 免费下载

例 10.23　森林

本例的森林包含了 3 棵树，如图 10.25 所示。

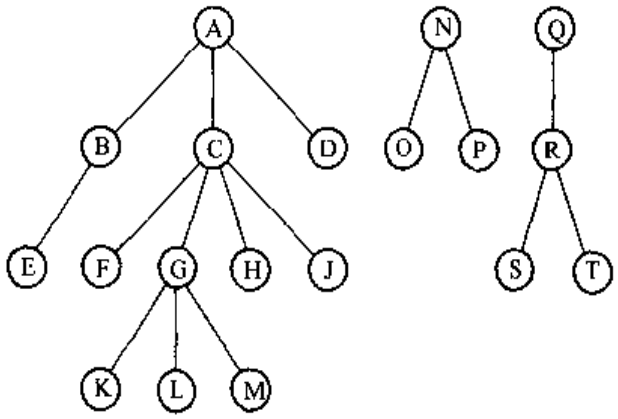


图 10.25　森林

下面的算法给出如何通过一棵二叉树来表示森林。

算法 10.8　从森林到二叉树的自然映射

1. 将森林中第一棵树的树根映射为二叉树的树根。
2. 如果节点 X 被映射为 X'，节点 Y 是 X 的第一个孩子，则将 Y 映射为 X'的左孩子。
3. 如果节点 X 被映射为 X'，节点 Z 是 X 的兄弟，则将 Z 映射为 X'的右孩子。森林中各树的树根间的关系视为兄弟关系。

例 10.24　从森林到二叉树的映射

本例将例 10.23 中的森林映射为二叉树，如图 10.26 所示。

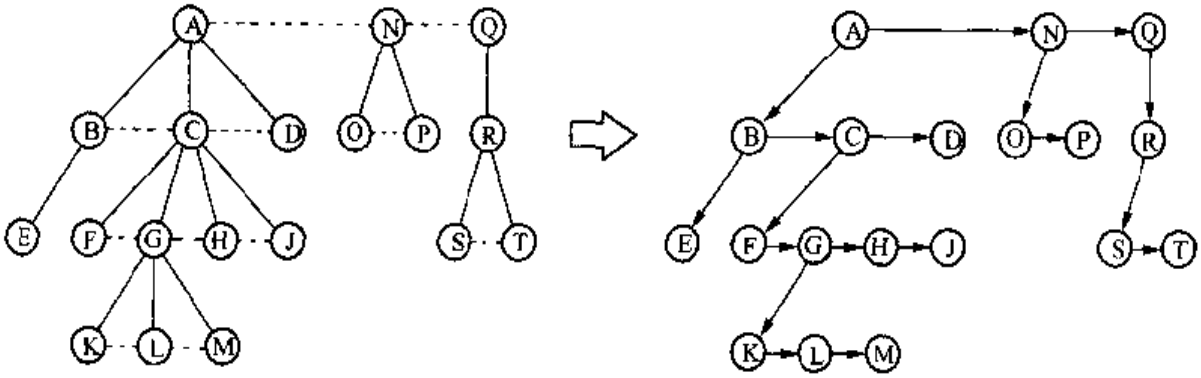


图 10.26　从森林到二叉树的映射

例如，在原来的森林中，C 的最左边的孩子是 F，兄弟是 D，因此，在二叉树中，C 的左孩子是 F，右孩子是 D。

复 习 题

10.1 一棵高度为3的满二叉树有多少个叶节点?

10.2 一棵高度为3的满二叉树有多少个中间节点?

10.3 一棵高度为3的满二叉树共有多少个节点?

10.4 一棵高度为9的满二叉树有多少个叶节点?

10.5 一棵高度为9的满二叉树有多少个中间节点?

10.6 一棵高度为9的满二叉树共有多少个节点?

10.7 请给出一棵节点数 n=100 的二叉树可能的高度范围?

10.8 为什么对于一般树没有中序遍历?

10.9 判断正误。

a. 如果一棵二叉树的所有叶节点都处于同层，这棵二叉树是满二叉树。

b. 如果二叉树的节点数为 n，高度为 h，则有 $h \geq \lfloor \lg n \rfloor$。

c. 二叉树在深度 d 最多有 $2d$ 个节点。

d. 如果一棵二叉树的每个合法的子树都是满二叉树，则这棵二叉树本身是满二叉树。

习 题

10.1 对于图 10.27 所示的这些二叉树，请按照二叉树的第二种定义（即每个内部节点都包含两个孩子）画出同它们等价的形式。

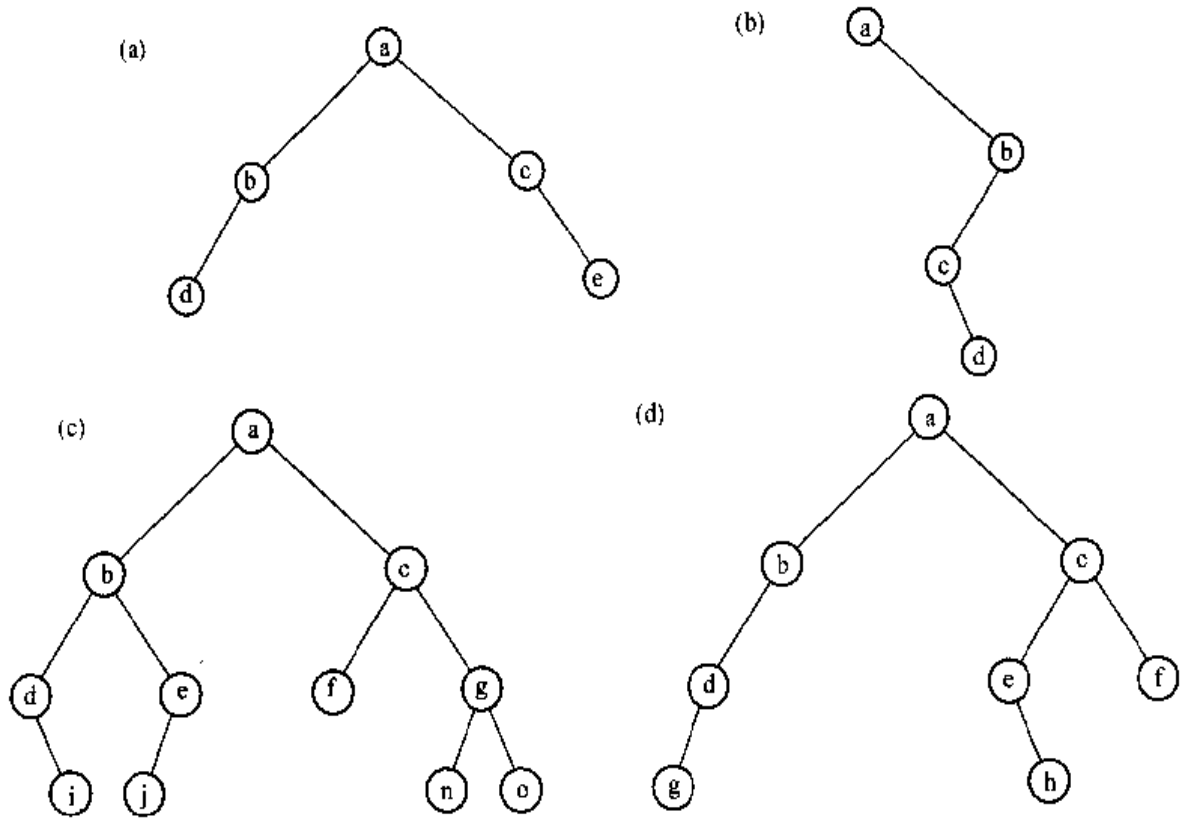


图 10.27 习题 10.1 图

标准分享网 www.bzfxw.com 免费下载

10.2　分别基于以下遍历算法给出如图 10.28 所示的二叉树的节点访问次序。

a. 分层遍历。

b. 前序遍历。

c. 中序遍历。

d. 后序遍历。

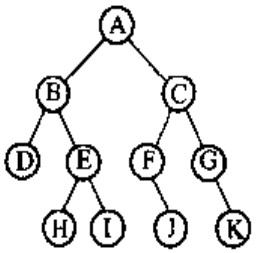


图 10.28　习题 10.2 图

10.3　分别基于以下遍历算法给出例 10.1 中大小为 10 的二叉树的节点访问次序。

a. 分层遍历。

b. 前序遍历。

c. 中序遍历。

d. 后序遍历。

10.4　分别基于以下遍历算法给出图 10.29 中二叉树的节点访问次序。

a. 分层遍历。

b. 前序遍历。

c. 中序遍历。

d. 后序遍历。

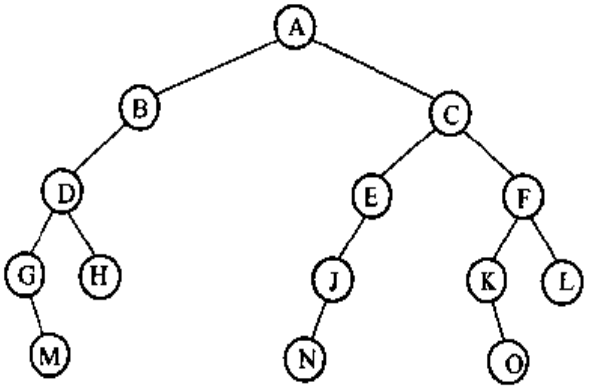


图 10.29　习题 10.4 图

10.5　按照二叉树的数组映射方法，给出习题 10.1 的二叉树的数组表示。

10.6　按照二叉树的数组映射方法，给出例 10.1 中二叉树的数组表示。

10.7　按照二叉树的数组映射方法，给出习题 10.4 中二叉树的数组表示。

10.8　假设二叉树节点的编号是按照数组映射进行的，再假设遍历算法中对节点的访问是将节点的名称打印出来，试问哪种遍历算法能够依次序将节点打印出来？

10.9　试画出表达式 $a*(b+c)*(d*e+f)$ 对应的表达式树。

10.10　试写出习题 10.9 中表达式对应的前缀及后缀表达式。

10.11　试画出习题 6.2 中各前缀表达式对应的表达式树。

10.12　试画出习题 6.4 中各中缀表达式对应的表达式树。

10.13　试画出习题 6.6 中各后缀表达式对应的表达式树。

10.14　试画出表达式 $a*(b+c)*(d*e+f)$ 对应的表达式树。

10.15　高度为 4 的二叉树的节点数 n 所处的区间是什么？

10.16　节点数为 7 的二叉树的高度 h 所处的区间是什么？

10.17　给定二叉树的节点数，要使树高最大，树应该是什么形状？

10.18 给定二叉树的节点数，要使树高最小，树应该是什么形状？

10.19 试用二叉树的递归定义验证图 10.30 中的二叉树？

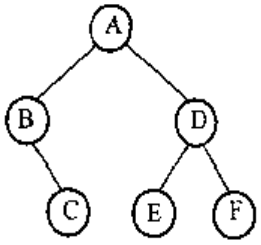


图 10.30 习题 10.19 图

10.20 试画出大小 n=5 的所有 42 棵二叉树。

10.21 大小 n=6 的不同二叉树有多少棵？

10.22 如果用 $f(n)$表示大小为 n 的二叉树的数目，试推导出 $f(n)$的循环关系。

10.23 试验证，当 $n \leqslant 8$ 时，由习题 10.22 导出的循环关系与下列公式结果相同：

$$f(n)=\frac{\binom{2n}{n}}{n+1}=\frac{(2n)!}{n!(n+1)!}\cdot\frac{(2n)(2n-1)(2n-2)...(2n+3)(2n+2)}{(n)(n-1)(n-2)...(2)(1)}$$

例如

$$f(4)=\frac{\binom{8}{4}}{5}=\frac{8!}{4!5!}\cdot\frac{(8)(7)(6)}{(4)(3)(2)(1)}=\frac{(8)(7)}{4}=14$$

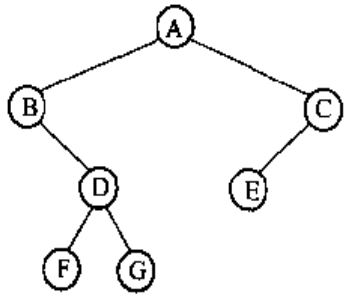


图 10.31 习题 10.26 图

10.24 证明推论 10.3。

10.25 证明定理 10.2。

10.26 画出如图 10.31 所示的二叉树对应的森林。

10.27 给出高度为 h 的完全二叉树的数目 $f(h)$的计算公式。

10.28 给出高度为 h 的满二叉树的数目 $f(h)$的计算公式。

10.29 证明满二叉树的每棵子树也都是满二叉树。

10.30 证明完全二叉树的每棵子树也都是完全二叉树。

类 BinaryTree 实现下列方法：

10.31
```
public int leaves();
// 结果返回树的叶节点个数
```

10.32
```
public int height();
// 结果返回树的高度
```

10.33
```
public int level();
// 如果指定的对象不在树中，结果返回-1；否则结果返回该对象在树中所处的层次
```

10.34
```
public void reflect();
// 将树中每个节点的孩子对换位置
```

10.35
```
public void defoliate();
// 将树中所有的叶节点移走
```

复习题答案

10.1 一棵高度为 3 的满二叉树有 $l=2^3=8$ 个叶节点？

标准分享网 www.bzfxw.com 免费下载

10.2 一棵高度为3的满二叉树有 $m = 2^3 - 1 = 7$ 个中间节点?

10.3 一棵高度为3的满二叉树共有 $n = 2^{3+1} - 1 = 2^4 - 1 = 15$ 个节点?

10.4 一棵高度为9的满二叉树有 $l = 2^9 = 512$ 个叶节点?

10.5 一棵高度为9的满二叉树有 $l = 2^9 - 1 = 512 - 1 = 511$ 个中间节点?

10.6 一棵高度为9的满二叉树共有 $n = 2^{9+1} - 1 = 2^{10} - 1 = 1023$ 个节点?

10.7 根据推论10.3，对于任意二叉树有$\lfloor \lg n \rfloor \leqslant h \leqslant n-1$，因此，一棵节点数 n=100的二叉树可能的高度范围为$\lfloor \lg 100 \rfloor \leqslant h \leqslant 100-1 = 99$。由于$\lfloor \lg 100 \rfloor = \lfloor 6.6 \rfloor = 6$。因此树高可能范围为 $6 \leqslant h \leqslant 99$。

10.8 二叉树的中序遍历算法在访问左子树和右子树之间递归当前子树的树根。该算法的前提是每个（非空）节点都恰好有两个（可能为空的）子树。对于一般的树，节点可以有任意数目的子树，因此，没有简单的中序遍历算法来遍历一般的树。

10.9 判断正误。

a. 真。

b. 真。

c. 真。

d. 假。

习 题 解 答

10.1 等价的形式如图10.32所示。

10.2 二叉树的节点访问次序如下：

a. 分层遍历：A，B，C，D，E，F，G，H，I，J，K

b. 前序遍历：A，B，D，E，H，I，C，F，J，G，K

c. 中序遍历：D，B，H，E，I，A，F，J，C，G，K

d. 后序遍历：D，H，I，E，B，J，F，K，G，C，A

10.3 二叉树的节点访问次序如下：

a. 分层遍历：A，B，C，D，E，F，H，I，J，M

b. 前序遍历：A，B，D，H，I，E，J，C，F，M

c. 中序遍历：H，D，I，B，J，E，A，F，M，C

d. 后序遍历：H，I，D，J，E，B，M，F，C，A

10.4 分别基于以下遍历算法给出二叉树的节点访问次序。

a. 分层遍历：A，B，C，D，E，F，G，H，J，K，L，M，N，O

b. 前序遍历：A，B，D，G，M，H，C，E，J，N，F，K，O，L

c. 中序遍历：G，M，D，H，B，A，N，J，E，C，K，O，F，L

d. 后序遍历：M，G，H，D，B，N，J，E，O，K，L，F，C，A

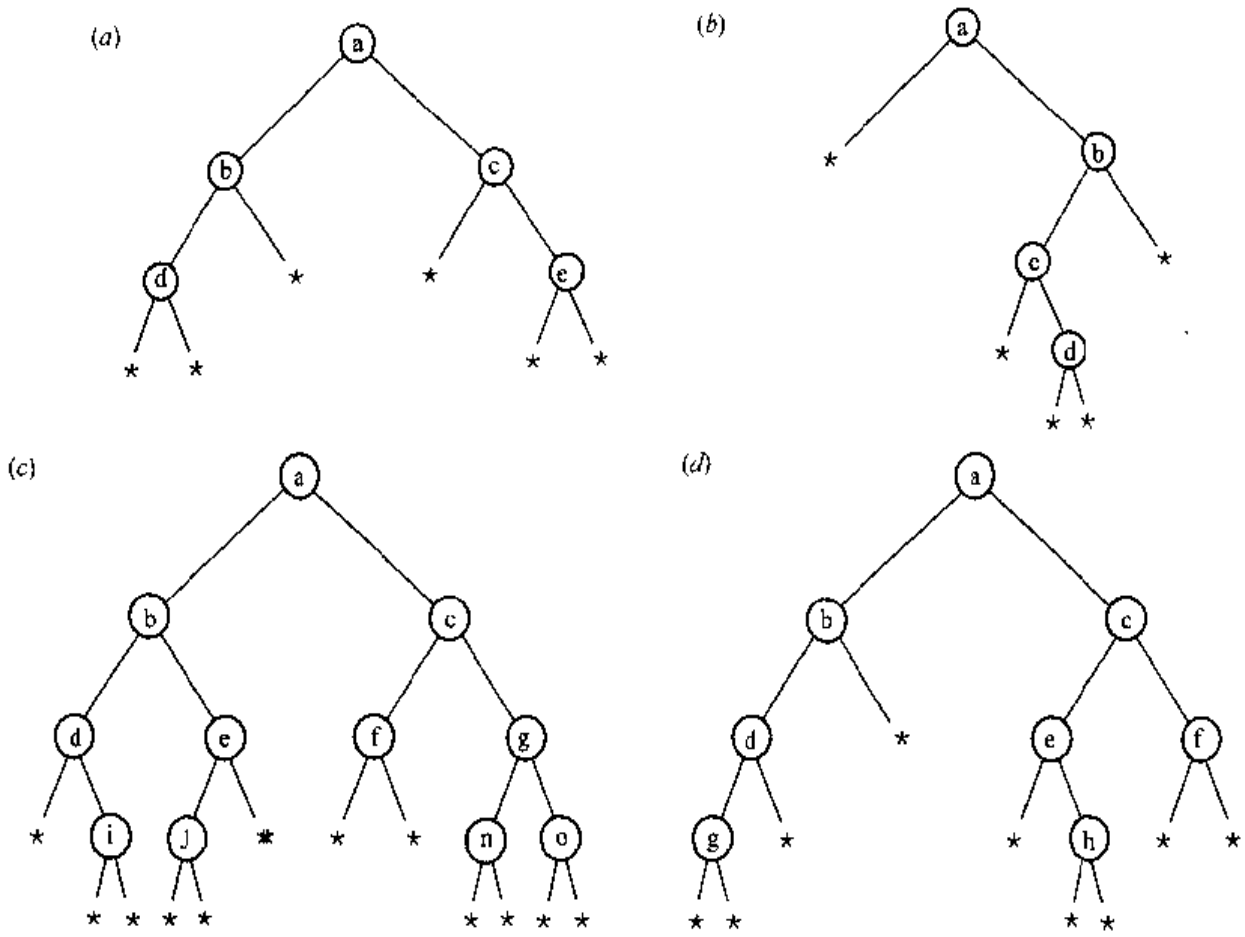


图 10.32 习题 10.1 答案

10.5 二叉树的数组映射如图 10.33 所示。

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	A	B	C	D	E	F	G			H	I		J		K

图 10.33 习题 10.5 答案

10.6 例 10.1 中二叉树的数组映射如图 10.34 所示。

0	1	2	3	4	5	6	7	8	9	10	11	12	13
	A	B	C	D	E	F		H	I	J			M

图 10.34 习题 10.6 答案

10.7 二叉树的数组映射如图 10.35 所示。

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
	A	B	C	D		E	F	G	H			J		K	L		M							N					O

图 10.35 习题 10.7 答案

10.8 分层遍历算法能够依次序将节点打印出来。

标准分享网 www.bzfxw.com 免费下载

10.9　表达式 $a*(b+c)*(d*e+f)$ 对应的表达式树如图 10.36 所示。

10.10　前缀表达式为 **a*+bc+*def*，后缀表达式为 **abc+de*f+***。

10.11　习题 6.2 中各前缀表达式对应的表达式树如图 10.37 所示。

10.12　习题 6.4 中各中缀表达式对应的表达式树如图 10.38 所示。

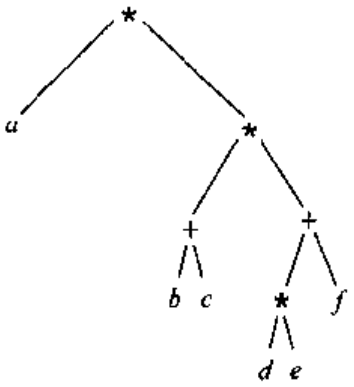


图 10.36　习题 10.9 答案

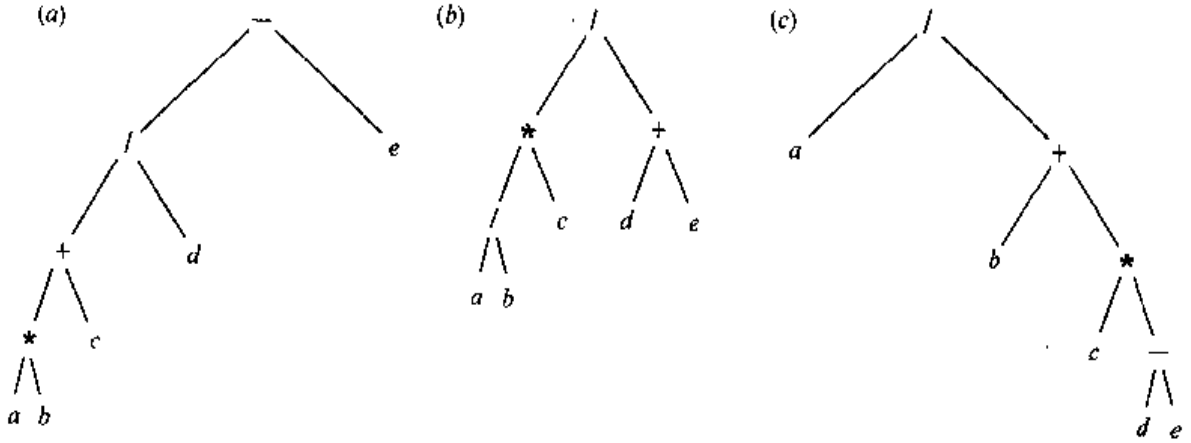


图 10.37　习题 10.11 答案

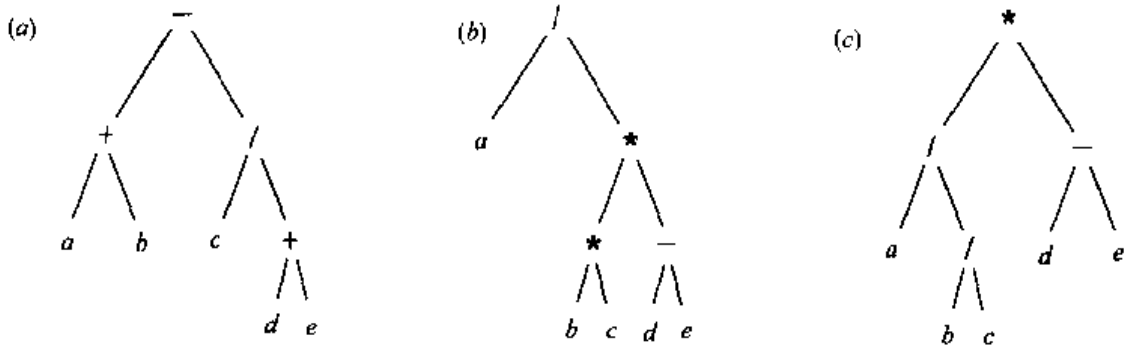


图 10.38　习题 10.12 答案

10.13　习题 6.6 中各后缀表达式对应的表达式树如图 10.39 所示。

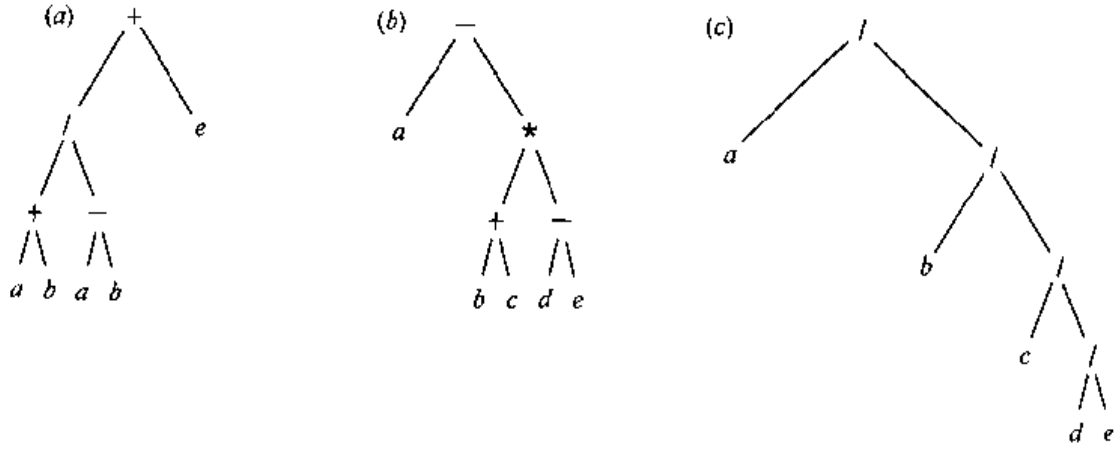


图 10.39　习题 10.13 答案

10.14 表达式 $a*(b+c)*(d*e+f)$对应的表达式树如图 10.40 所示。

10.15 高度为 4 的二叉树的节点数 n 所处的区间是 $5 \leqslant n \leqslant 31$。

10.16 节点数为 7 的二叉树的高度 h 所处的区间是 $2 \leqslant h \leqslant 6$。

10.17 给定二叉树的节点数，要使树高最大，树应该是线性序列。

10.18 给定二叉树的节点数，要使树高最小，树应该是完全二叉树。

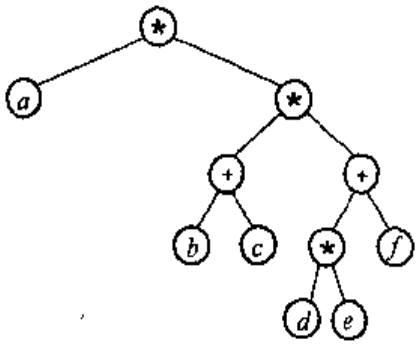


图 10.40 习题 10.14 答案

10.19 为了用二叉树的递归定义验证给定二叉树，我们首先考察叶节点 C、E 和 F。它们都是单个二叉树，左子树和右子树都为空，满足二叉树的递归定义。接下来，以节点 B 为根的子树是二叉树，因为它是一个三元组（X，L，R）其中 X=B，L=Φ，R=C。类似，以节点 D 为根的子树是二叉树，因为它是一个三元组（X，L，R）其中 X=D，L=E，R=F。最后，整个树是二叉树，因为它是一个三元组（X，L，R）其中 X=A，L 是以 B 为根的二叉树，R 是以 D 为根的二叉树。

10.20 图 10.41 是大小 n=5 的所有 42 棵二叉树。

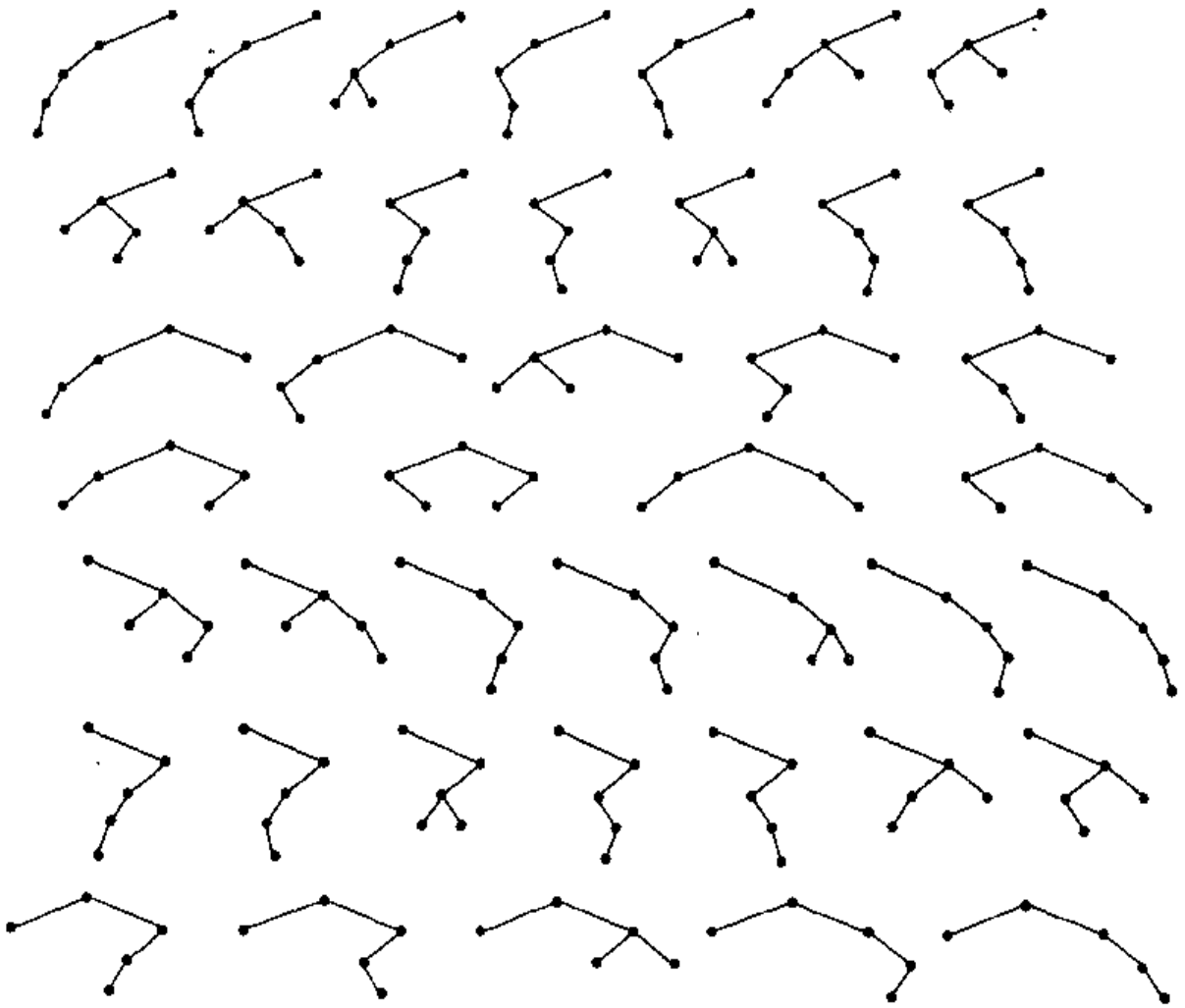


图 10.41 习题 10.20 答案

标准分享网 www.bzfxw.com 免费下载

10.21　大小 n=6 的不同二叉树有 132 棵：1×42+1×14+2×5+5×2+14×1+42×1=132。

10.22　一棵非空二叉树包括树根 X，左子树 L，右子树 R。令 n 为该二叉树的大小，令 n_L=|L|=L 的大小，n_R=|R|=R 的大小。则 n=1+L+R。对于（n_L，n_R），共有 n 种不同的可能：（0, n–1），（1, n–2），…，（n–1, 0）。例如，如果 n=6，则共有 6 种不同的可能：（0, 5），（1, 4），（2, 3），（3, 2），（4, 1），（5, 0）。对于（0, n–1）的情况，L 是空子树，|R|为 n–1；这种情况共有 $f(0)\times f(n-1)$种不同的二叉树。对于（1, n–2）的情况，L 是单节点树，|R|为 n–2；这种情况共有 $f(1)\times f(n-2)$种不同的二叉树。依此类推，大小为 n 的不同二叉树的数量为：

$f(n) = 1\times f(n-1) + 1\times f(n-2) + 2\times f(n-3) + 5\times f(n-4)\ 14\times f(n-5) + \ldots + f(i-1)\ \times f(n-i) + f(n-1)\ \times 1$，即

$$f(n)=\sum_{i=1}^{n} f(i-1)\cdot f(n-i)$$

10.23　这个数组被称为 *Catalan* 数，如表 10.1 所示。

表 10.1　Catalan 数表

n	$\binom{2n}{n}$	n+1	$\binom{2n}{n}/(n+1)$	$\sum f(i-1)\cdot f(n-i)$
0	1	1	1	1
1	2	2	1	1 • 1 = 1
2	6	3	2	1 • 1+1 • 1 = 2
3	20	4	5	1 • 2+1 • 1+2 • 1 = 5
4	70	5	14	1 • 5+1 • 2+2 • 1+5 • 1 = 14
5	252	6	42	1 • 14+1 • 5+2 • 2+5 • 1+14 • 1 = 42
6	924	7	132	1 • 42+1 • 14+2 • 5+5 • 2+14 • 1+42 • 1 = 132
7	3432	8	429	1 • 132+1 • 42+2 • 14+5 • 5+14 • 2+42 • 1+132 • 1 = 429

10.24　对于给定高度 h>0，具有最多节点的二叉树是满二叉树。推论 10.2 说明节点数为 $n=2^{h+1}-1$。因此，任何高度为 h 的二叉树，其节点数 n 必须满足 $n\leqslant 2^{h+1}-1$。对于给定高度 h，具有最少节点的二叉树的每个内部节点都只有一个孩子。这棵线性树的节点数为 $n=h+1$，因为除了惟一的叶节点外，每个内部节点都有且仅有一个孩子。因此，对于给定高度 h 的二叉树，其节点数最少必须有 h+1。

10.25　令 T 是一棵高度为 h 大小为 n 的完全二叉树。令 T_1 是由 T 去掉最下层叶节点而得到的满的子树。T_2 是由 T 补齐最下层叶节点而得到的满的超树。因此，T_1 的高度为 h–1，T_2 的高度为 h。根据推论 10.1，$n_1=|T_1|=2^h-1$，$n_2=|T_2|=2^{h+1}-1$。现有 $n_1<n\leqslant n_2$，因此 $2^h-1=n_1<n\leqslant n_2=2^{h+1}-1$，则有 $2^h=n_1+1\leqslant n\leqslant n_2<n_2+1=2^{h+1}$。因而有 $h\leqslant \lg n<h+1$，即 $h=\lfloor \log_2 n \rfloor$。

10.26　根据自然映射的逆映射，可以得到二叉树对应的森林如图 10.42 所示。

10.27　$f(h)$=1。

10.28　$f(h)$=h。

10.29　定理：满二叉树的每棵子树也都是满二叉树。

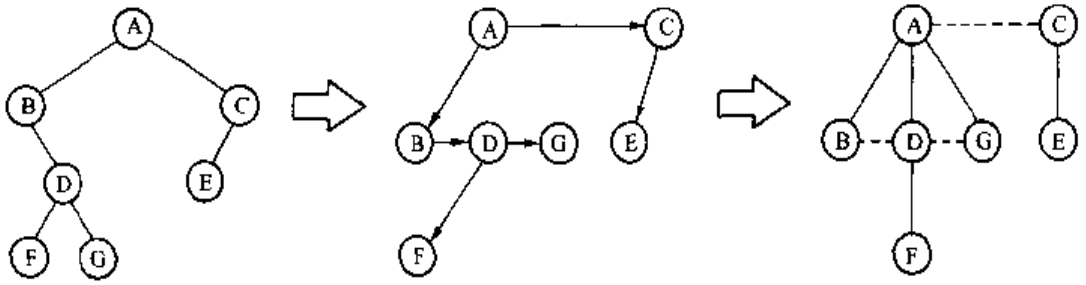


图 10.42 习题 10.26 答案

证明：令 T 是二叉搜索树，S 为 T 的子树。令 x 是 S 中的任意元素，L 和 R 分别为 S 中 x 的左、右子树。因为 S 是 T 的子树，x 也是 T 的元素，L 和 R 也分别为 T 中 x 的左、右子树。由于 T 具备 BST 特性，因此，对于每个 $y\in L$，$z\in R$ 有 $y\leqslant x\leqslant z$。同样 S 也具备 BST 特性。

10.30 定理：完全二叉树的每棵子树也都是完全二叉树。

证明：令 T 是二叉搜索树，S 为 T 的子树。令 x 是 S 中的任意元素，L 和 R 分别为 S 中 x 的左、右子树。因为 S 是 T 的子树，x 也是 T 的元素，L 和 R 也分别为 T 中 x 的左、右子树。由于 T 具备 BST 特性，因此，对于每个 $y\in L$，$z\in R$ 有 $y\leqslant x\leqslant z$。同样 S 也具备 BST 特性。

10.31
```
public int leaves()
{  if (this = = null) return 0;
   int leftLeaves = (left= = null ? 0 : left.leaves());
   int rightLeaves = (right= = null ? 0 : right.leaves());
   return leftLeaves + rightLeaves;
}
```

10.32
```
public int height()
{  if (this = = null) return -1;
    int leftHeight = (left== null ?  -1 : left.height());
    int rightHeight = (right==null ? -1 : right.height());
    return 1 + (leftHeight<rightHeight ? rightHeight : leftHeight);
}
```

10.33
```
public int level; (Object object)
{  if (this = = null) return -1;
   if (object = = root) return 0;
   int leftLevel = (left= =null ? -1 : left.level (object));
   int rightLevel = (right= =null ? -1 : right.level (object));
   if (leftLevel<0 && rightLevel<0) return -1;
   return 1 + (leftLevel<rightLevel ? rightLevel : leftLevel);
 }
```

10.34
```
public void reflect()
```

标准分享网 www.bzfxw.com 免费下载

```
{  if (this = = null) return;
   if (left != null) left.reflect();
   if (right != null) right.reflect();
   BinaryTree temp=left;
   left = right;
   right = temp;
```

10.35 `public void defoliate()`

```
{  if (this = = null) return;
   if (left = = null && right = = null)
{  root = null;
   return;
}
   if (left != null && left.left= =null && left.right= =null)
   left = null;
   else left.defoliate();
   if (right != null && right.left= =null && right.right= =null)
   right = null;
   else right.defoliate();
}
```

第11章 搜 索 树

树结构常常用以存储数据，因为它们的组织结构能够提供对数据的有效访问。搜索树是对所存储的数据进行了某种排序的树结构。

11.1 多路搜索树

下面给出多路搜索树的递归定义：

所谓 m 阶多路搜索树是：

1) 空集；

2) 二元组 (k, S)，其中，序列 $k=(k_1, k_2, ..., k_{n-1})$ 是 $n-1$ 个键，序列 $S=(S_0, S_1, S_2, ..., S_{n-1})$ 是 n 个 m 阶多路搜索树，其中 $2 \leqslant n \leqslant m$，且对所有 $s_i \in S_i$ 有 $s_0 \leqslant k_1 \leqslant s_1 \leqslant \ldots \leqslant k_{n-1} \leqslant s_{n-1}$。

这个定义与一般树的定义（参见 9.1 节）类似。m 阶多路搜索树可以看成一种特殊的 m 阶树，只不过树的节点满足上面提到的序关系的键的序列。

例 11.1 一棵 5 路搜索树

图 11.1 给出一棵 5 路搜索树的例子。它包含 3 个度为 5 的中间节点（每个节点包含 4 个键），3 个度为 4 的中间节点（每个节点包含 3 个键），4 个度为 3 的中间节点（每个节点包含 2 个键）和 1 个度为 2 的中间节点（它包含 1 个键）。

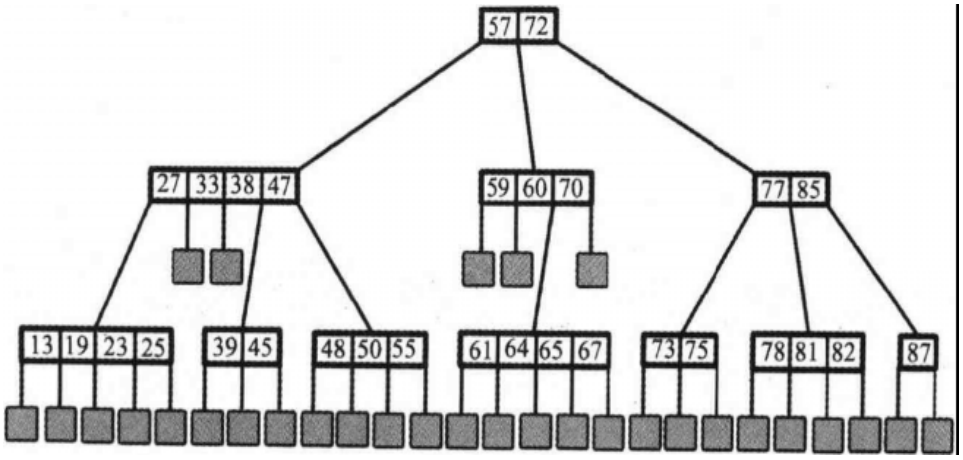


图 11.1 5 路搜索树

根节点包含 2 个键，共有 3 个孩子。其第一个孩子的所有键都小于 k_1=57。第二个孩子的所有键介于 k_1=57 和 k_2=72 之间。第三个孩子的所有键都大于 k_2=72。实际上，这棵搜索树的第一棵子树中所有 13 个键都小于 57，第二棵子树中所有 7 个键都介于 57 和 72 之间，第三棵子树中的所有 8 个键都大于 72。

标准分享网 www.bzfxw.com 免费下载

m 阶搜索树之所以被称为搜索树是因为它为大型列表的搜索提供了多层次索引。为了搜索一个键值，我们从树根开始向下搜索直到发现该值或到达某个叶节点。在每个节点，都根据键进行一个折半查找。如果在这个节点没有发现该值，则给出包含该值的两个相邻键值（$k_0=-\infty$，$k_n=+\infty$），然后根据这两个相邻键值引出的链接转向下一节点。如果搜索过程到达一个叶节点，则说明我们要找的键值不在搜索树中。

例如，为了搜索键值 66，从树根开始沿着中间的链接（因为 57≤66<72）下降到中间那个具有 3 个键值的节点。然后，再沿着其第三条链接（因为 60≤66<70），下降到下层那个具有 4 个键值的节点，最后沿着其第三条链接（因为 65≤66<67）下降到叶节点，这就说明键值 66 不在这棵搜索树中。

为了向 m 阶搜索树中插入一个键值，首先需要运用搜索算法。如果搜索算法在某个叶节点停止，那么其父节点中的两个作为上下界的键值就给出了待插入键值的位置信息。剩下的工作就是在上下界所指示的位置插入新键值了。如果本次插入动作使得当前节点包含 m 个键值了（即超过了每个节点最多 m–1 个键值的限制），则需要对当前节点进行拆分。拆分分为两步：首先确定当前节点的中间键值，然后把该键值提升到父节点中适当位置同时将当前节点在该键值处一分为二。如果键值的提升导致了父节点的键值数也超过限制，则按照同样的策略对父节点进行拆分。需要的话，这种拆分过程可以一直扩散到根节点，这时，将产生一个新的根节点，搜索树的高度也随着增加一层。

例 11.2　5 路搜索树的插入

本例说明如何将 66 插入到例 11.1 的搜索树中。首先按照前面描述的搜索过程搜索这棵搜索树，这一搜索过程最终将停留在图 11.2 中标有标记 X 的叶节点上。

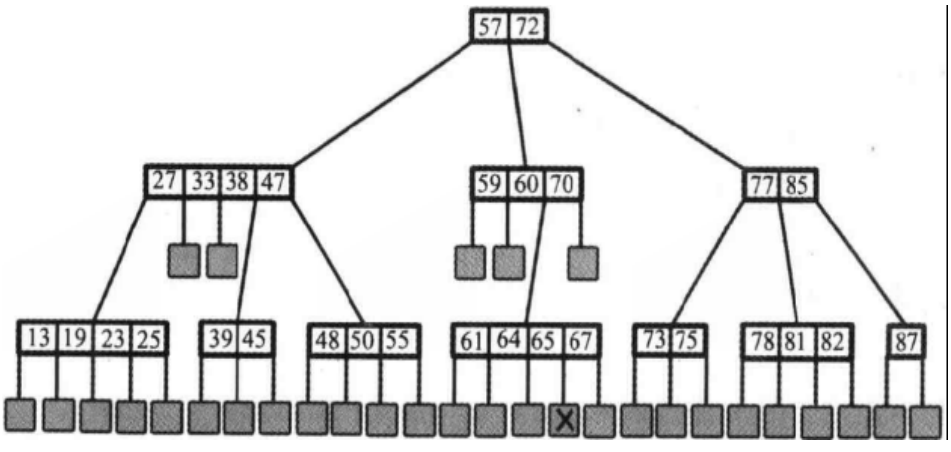


图 11.2　搜索树插入过程 1

将键值 66 插入到其父节点的键值 65 和 67 之间，如图 11.3 所示。

现在可以看到，键值 66 的插入使得当前节点包含 5 个键值，违反了 5 路搜索树最多包含 4 个键值的限制。因此，需要对该节点进行拆分，将中间键值 65 上移到父节点中，如图 11.4 所示：

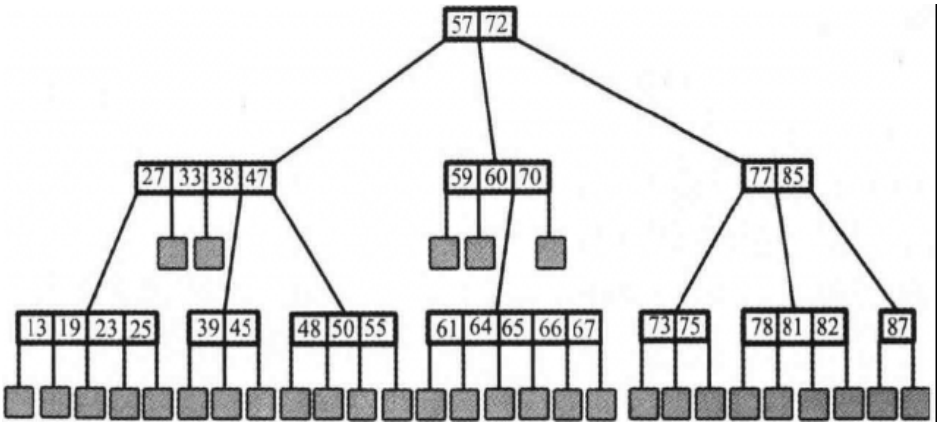


图 11.3 搜索树插入过程 2

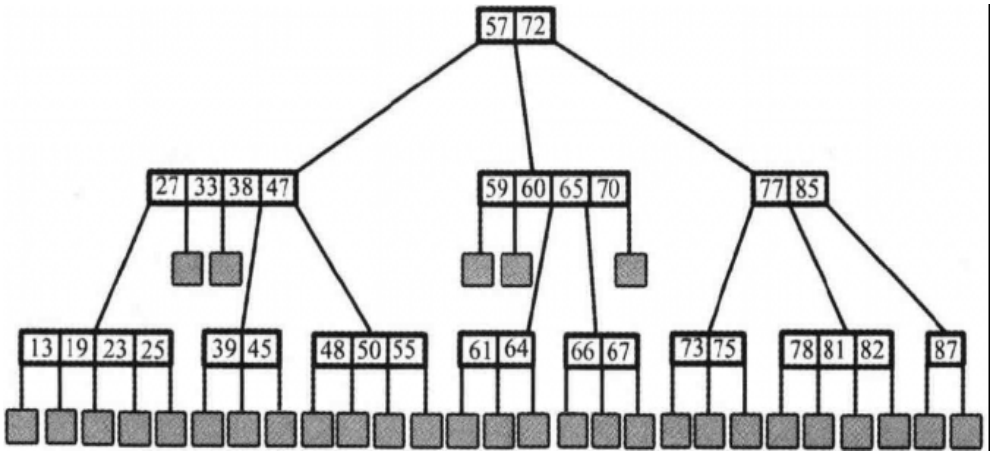


图 11.4 搜索树插入过程 3

节点拆分的情况出现的频率并不会很高，特别是当 m 较大时。比如，如果 m=50，则平均只会出现 2%的满节点。也就是说，对于插入操作而言，搜索树的最底层节点只有 2%的概率出现节点的拆分。倒数第二层节点的二次拆分概率就只有 2%的 2%，即 0.000 4。类似，3 次拆分概率就是 0.000 008。因此，根节点被拆分的概率就微乎其微，而根节点拆分是搜索树增高的前提。这样，搜索树就会保持一种矮而宽的形状，加快搜索时间。

11.2 B-树

m 阶 B-树首先是一棵 m 路搜索树，此外还需满足下列条件：

1. 根节点至少要有两个孩子；
2. 所有其他内部节点至少有$\lceil m/2 \rceil$个孩子；
3. 所有叶节点都处于同一层。

这些额外的条件一方面使得搜索树更加平衡（因而也有更高的效率），另一方面简化了插入和删除算法。

B-树一般用作磁盘上大量存储数据的索引。在关系数据库中，数据被组织成独立的记

标准分享网 www.bzfxw.com 免费下载

录序列，即表。每张表都被存储成顺序数据文件，就像是数组中的数据元素的结构一样。或者，关系数据库系统也可以通过磁盘地址直接访问那些记录。无论是哪种方法，每个记录都必须按照某种地址策略被直接访问。如果有了它们的磁盘地址，就可以对它们进行直接访问（只需一次磁盘读操作）。因此，在 B-树中存储的“键值”实际上是一个（键值，地址）对儿，其中包含了记录的实际主键值（比如人员记录的社会保险号，书籍记录的 ISBN 号）以及其磁盘地址。在下文中，为简化表述我们只提键值，但需要知道，实际上还有磁盘地址信息。

例 11.3 B-树

图 11.5 给出的是一棵 5 阶 B-树。它的内部节点有 3、4 或 5 个孩子，所有的叶节点都处于第 3 层。

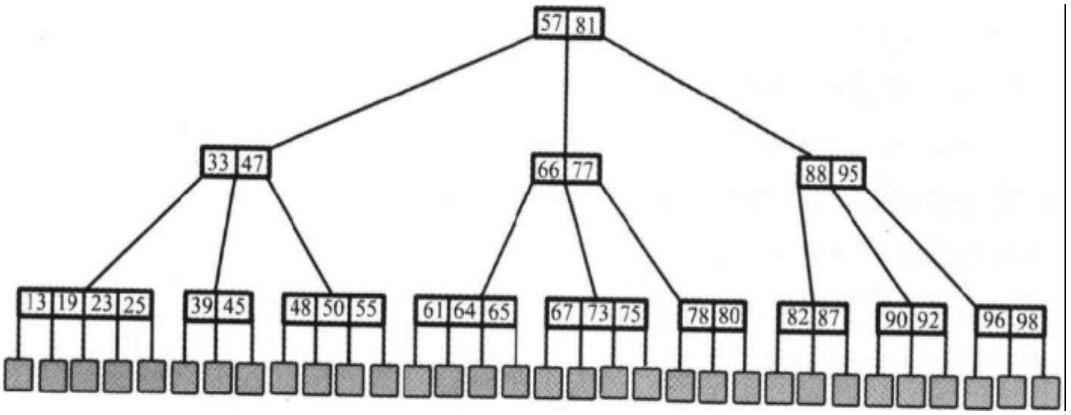


图 11.5 5 阶 B 树

算法 11.1 B-树的搜索

给定键值 k，为了在 m 阶 B-树的索引中定位对应记录：

1. 如果树为空，返回 null。
2. 令 x 为根节点。
3. 重复步骤 4~6，直至 x 为叶节点。
4. 在节点 x 使用折半查找算法（参见算法 2.2）搜索键值 k_i，其中，$k_{i-1}< k\leqslant k_i$ （$k_0=-\infty$，$k_m=+\infty$）。
5. 如果 $k_i=k$，从磁盘读取相应记录并返回之。
6. 令 x 为子树 S_i 的根节点。
7. 返回 null。

注意到，上面这个算法的过程实际上非常类似于我们读书时对索引的使用过程。索引中的每一页都通过一个词或字母来代表这一页的主题。这些词或字母就好比搜索树内部节点上的键值，而紧跟在词或字母后面的页码就好比数据文件的磁盘地址，按照页码翻到特定的页就好比从磁盘上定位文件。如果书籍的索引本身还有一个索引的话，这个类比就更加形象了。搜索树中不同的内部节点层分别对应着不同的索引级别。

算法 11.2 B-树的插入

给定键值 k，在 m 阶 B-树的索引中插入对应记录：

1. 如果树为空，创建一个带有两个哑叶节点的树根，并将 k 插入到根节点上，返回 `true`（表示插入操作成功）。
2. 令 x 为根节点。
3. 重复步骤 4~6，直至 x 为叶节点。
4. 在节点 x 使用折半查找算法（参见算法 2.2）搜索键值 k_i，其中，$k_{i-1}< k\leqslant k_i$ （$k_0=-\infty$，$k_m=+\infty$）。
5. 如果 $k_i=k$，返回 `false`（表示插入操作失败，因为已经存在一个以 k 为键值的记录了，而键值在标志应该惟一）。
6. 令 x 为子树 S_i 的根节点。
7. 将记录存入磁盘。
8. 在节点 x 的键值 k_{i-1} 和 k_i 之间插入 k（及记录的磁盘地址）。
9. 在节点 x 加入一个哑叶节点。
10. 如果 degree(x) = m，重复步骤 11~13 直到 degree(x) < m。
11. 令 k_j 为节点 x 的中间键值。
12. 从 x 中将 k_j 移出将使原来的键值分为两部分，令 u 和 v 分别为左、右部分。
13. 如果节点 x 是根节点，创建一个包含 k_j 并且以 u、v 为左、右子树的新根节点。
14. 否则，将 k_j 插入到 x 的父节点中并维护父节点同 u、v 的链接。
15. 返回 `ture`。

B-树的插入算法示例如图 11-6 所示。

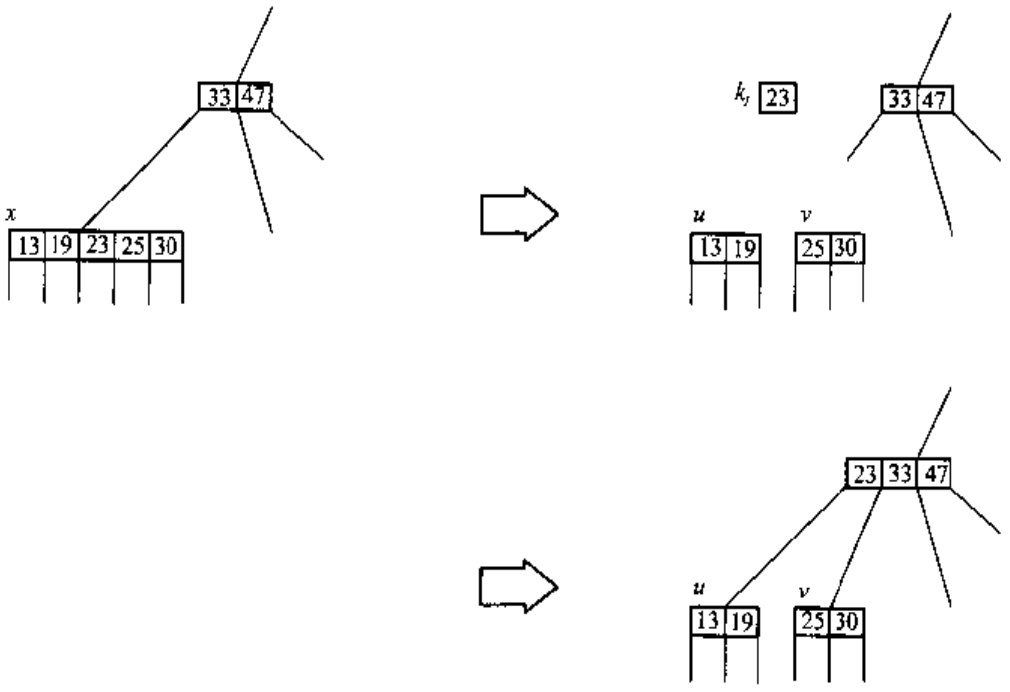


图 11-6 B-树插入算法示例

标准分享网 www.bzfxw.com 免费下载

B-树的删除算法与插入算法类似。

前面介绍的这 3 个算法的执行时间都与树高成正比。从推论 9.1 可知，树高同 $\log_m n$ 成正比。再由定理 A.6 有，算法执行时间与 $\lg n$ 成正比。因此，我们有：

定理 11.1 在 B-树中，搜索、插入和删除都能够在 $O(\lg n)$时间完成

11.3 二叉搜索树

二叉搜索树是一棵元素包含某种类型的键的二叉树，它还要满足下列性质：假设 k 是任意节点中的键值，则该节点的左子树中的任意节点的键值都小于或等于 k，右子树中的任意节点的键值都大于或等于 k。这一性质称为 BST 性，它保证了按照中序遍历对二叉搜索树进行遍历时得到的递增序列，如图 11.7 所示。

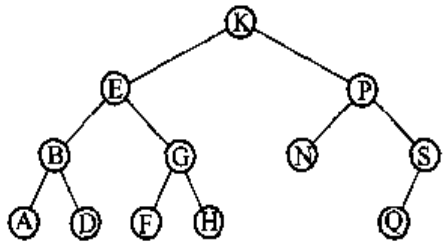


图 11.7 二叉搜索树

在向二叉搜索树中插入元素时，也要保证 BST 性：

算法 11.3 二叉搜索树的插入

给定键值 k，为了在二叉搜索树中插入元素的步骤：

1. 如果树为空，创建树根并在树根插入新元素，返回。
2. 令 p 为树根。
3. 如果键值 k 小于 p 中的值且 p 没有左孩子，将新元素作为 p 的左孩子，返回。
4. 如果键值 k 小于 p 中的值且 p 有左孩子，重新设置 p 为其左孩子，转步骤 3。
5. 如果节点 p 没有右孩子，将新元素作为 p 的右孩子，返回。
6. 重新设置 p 为其右孩子，转步骤 3。

例 11.4 二叉搜索树的插入

在本例中（如图 11.8 所示），我们将利用算法 11.3 将键值为 M 的元素插入到上图中的二叉搜索树中。

在步骤 1，叠代器 p 定位在根节点 K。由于 M 大于 K（根据字母序）且 K 有右孩子，算法进入步骤 6，叠代器 p 重新定位在节点 P，再转回步骤 3 继续。由于 M 小于 P 且 P 有左孩子，算法进入步骤 4，叠代器 p 重新定位到节点 N，再转回步骤 3 继续。由于 M 小于 N 但是 N 没有左孩子，算法进入步骤 5，将新元素作为 N 的左孩子插入到树中，返回。

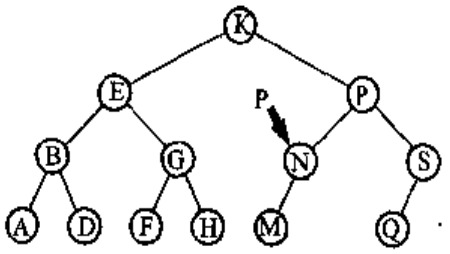


图 11.8 二叉搜索树的插入

例 11.5 构造二叉树

图 11.9 说明的是如何通过插入序列 44，77，55，22，99，33，88 来构造二叉搜索树：

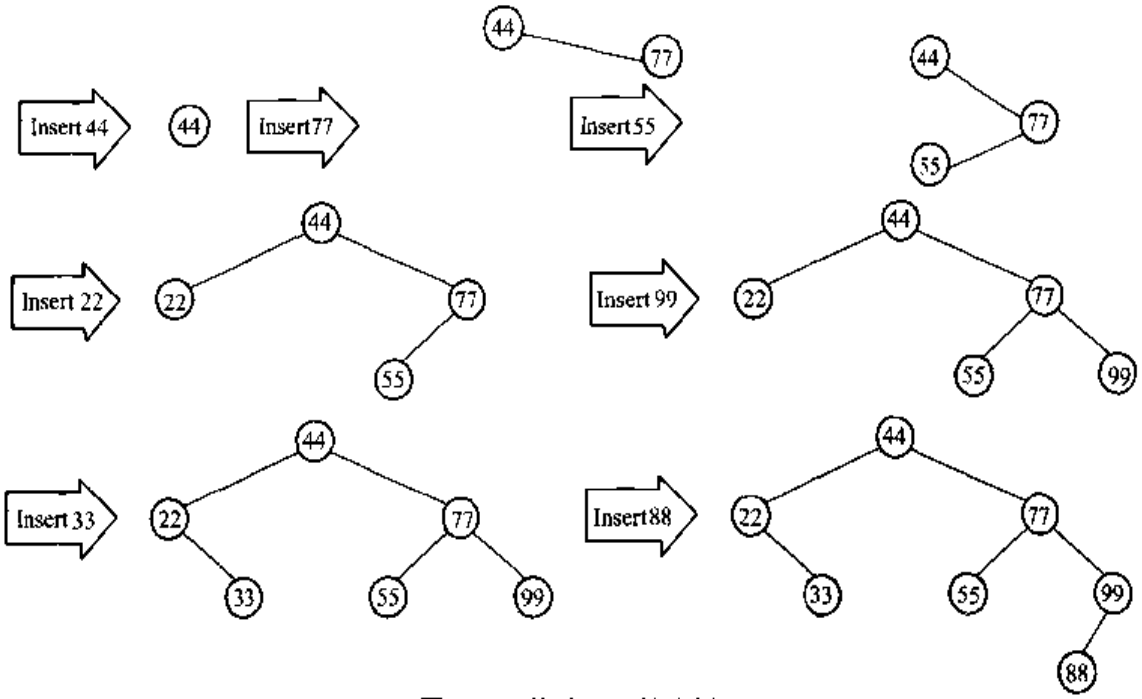


图 11.9 构造二叉搜索树

二叉搜索树如果是平衡的，它的搜索效率也会很高。因为对于折半查找算法而言，需要花费 $O(\lg n)$步就可以从一棵平衡二叉搜索树中定位一个元素。但是，如果没有更进一步的限制，二叉搜索树可能是非常不平衡的。最坏的情况是当我们按照顺序插入元素时，二叉搜索树就会蜕变成一个线性表，从而使搜索时间变成 $O(n)$的顺序搜索。

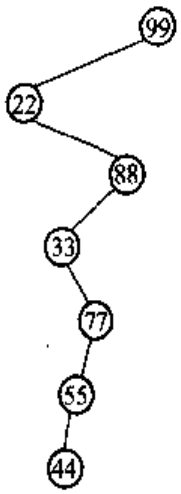


图 11.10 不平衡二叉搜索树

例 11.6 不平衡二叉搜索树

本例中的输入时间同例 11.5 的相同但是插入的顺序不同，为：99，22，88，33，77，55，44。最终得到的二叉搜索树如图 11.10 所示。

这表明同样的数据的不同插入顺序会导致不同的树形。更为重要的是，它还表明二叉搜索树有可能蜕变成线性的或是近线性的。

11.4 二叉搜索树的性能特点

二叉搜索树的 insert()和 search()操作都是从树根开始向下进行的，在每层上进行一次比较。因此，这两个算法的执行时间都正比于 $h+1$，其中 h 是树高。虽然 search()操作可能在到达叶节点之前就会终止，但是 $h+1$ 仍然是比较次数的上界。

定理 11.2 在大小为 n 的二叉搜索树中，insert()和 search()操作在最好的情况下需要 $O(\lg n)$次比较。

证明：在最好的情况下，二叉树是完全、平衡的和接近于满的。根据定理 11.3 有 $h+1$

标准分享网 www.bzfxw.com 免费下载

$\approx \lg(n+1) = O(\lg n)$。

推论 11.1　在大小为 n 的二叉搜索树中，insert()和 search()操作在最坏的情况下需要 $O(n)$次比较。

证明：在最坏的情况下，树蜕变成线性的，所以 $h+1 = n = O(n)$。

定理 11.3　在大小为 n 的二叉搜索树中，insert()和 search()操作在平均的情况下需要 $O(2\ln n) = O(1.39\lg n)$次比较。

此定理的证明超出本书范围。

11.5　AVL 树

在例 11.6 中表现出来的二叉搜索树的不平衡问题，可以通过在节点上加入平衡限制条件加以避免。我们将任意节点的左右子树的高度差定义为该节点的“平衡数”。通过引入这个概念，我们可以定义出所谓 AVL 树：任意节点的平衡数都为–1、0 或 1 的二叉搜索树。它是根据它的两个发明者的名字（G. M. Adel'son-Velskii 及 Y. M. Landis）命名的。

图 11.11 给出的树不是 AVL 树，因为节点 C 是不平衡的，其平衡数是 2，超过了 AVL 树对平衡数的限制。同样，在节点 G 也是不平衡的。图 11.12 所示的树是 AVL 树，因为其任意节点的平衡数都是–1、0 或 1。

AVL 树与斐波那契数列 F_m 有密切的关系（参见 A.10 节）。

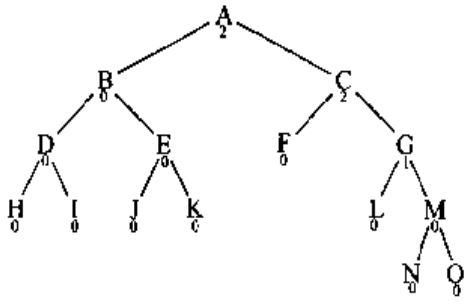


图 11.11　非 AVL 树

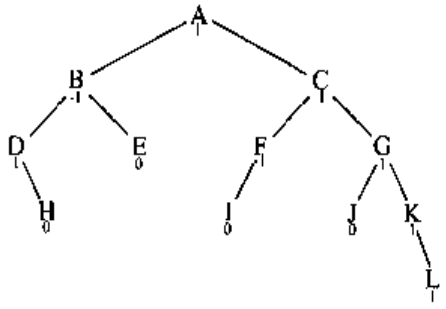


图 11.12　AVL 树

定理 11.4　如果一棵 AVL 树的大小为 n 高度为 h，则 $n \geqslant F_{h+2} - 1$

证明：设 T 是高度为 h 大小为 n 的 AVL 树。如果 $h = -1$ 则 $n=0$（即空树），$F_{h+2} - 1 = F_{(0)+2} - 1 = F_2 - 1 = 1 - 1 = 0 = n$。如果 $h=0$ 则 $n=1$（即树只有一个节点），$F_{h+2} - 1 = F_{(1)+2} - 1 = F_3 - 1 = 2 - 1 = 1 = n$。这构成我们递归证明的基础。

现在令 $h>1$ 并假设（归纳假设）不等式对所有高度小于 h 的 AVL 树都成立。令 T 是高度为 h 大小为 n 的 AVL 树，则 T 的根节点的平衡数是–1、0 或 1。令 T_L 和 T_R 分别是 T 的左右子树，h_L 和 h_R 分别是它们的高度，n_L 和 n_R 分别是它们的大小。则 $h_L = h-1$ 或者 $h_R = h-1$ 且 $h_l \geqslant h-2$ 及 $h_R \geqslant h-2$。无论哪种情况，$F_{h_R} + F_{h_L} \geqslant F_{h-1} + F_{h-2} = F_h$。因此，根据归纳假设，$n = 1 + n_L + n_R \geqslant 1 + (F_{h_L+2} - 1) + (F_{h_R+2} - 1) \geqslant F_{h+2} - 1$。

推论 11.2 AVL 树的高度是 $O(1.44 \lg n)$的。

证明：根据推论 A.3，$F_{h+2} = \Theta(\phi^{h+2}) = \Omega(\phi^h)$。因此有 $n \geqslant F_{h+2} - 1 = \Omega(F_{h+2}) = \Omega(\phi^h)$；即 $n = \Omega(\phi^h)$。因此，$\phi^h = O(n)$，$h = O(\log_\phi n) = O(1.44 \lg n)$。

推论 11.3 在 AVL 树中，搜索、插入和删除操作的执行时间最多比完全平衡树的慢 44%。

11.6 类 AVLTree

AVL 树的类扩展了 10.8 节定义的类 BinaryTree：

```
public class AVLTree extends BinaryTree
{ protected AVLTree left, right;
  protected int balance;
  protected java.util.Comparator comp;

  public AVLTree(java.util.Comparator comp)
  { this.comp = comp;
  }

  public AVLTree(Object root, java.util.Comparator comp)
  { this.root = root;
    this.comp = comp;
  }
  public boolean add(Object object)
  { AVLTree temp = attach(object);
    if (temp != this)
    { left = temp.left;
      right = temp.right;
      balance = temp.balance;
    }
    return true;
  }

  public AVLTree attach(Object object)
  { if (root == null)  // tree is empty

    { root = object;
      return this;
    }

    if (comp.compare(object,root)<0)  // insert into left subtree
    { if (left == null)
      { left = new AVLTree(object,comp);
        ++size;
```

标准分享网 www.bzfxw.com 免费下载

```
        --balance;
      }
      else
      { int lb = left.balance;
        left = left.attach(object);
        if (left.balance != lb && left.balance != 0) --balance;
      }
      if (balance < -1)
      { if (left.balance > 0) left = left.rotateLeft();
        return rotateRight();
      }
    }
    else // insert into right subtree
    { if (right == null)
      { right = new AVLTree(object,comp);
        ++size;
        ++balance;
      }
      else
      { int rb = right.balance;
        right = right.attach(object);
        if (right.balance != rb && right.balance != 0) ++balance;
      }
      if (balance > 1)
      { if (right.balance < 0) right = right.rotateRight();
        return rotateLeft();
      }
    }
    return this;
  }
  private AVLTree rotateRight();
  // 参见习题11.6
    private AVLTree rotateLeft()
  { // PRECONDITION: this.right != null;
    AVLTree x = this;
    AVLTree y = right;
    AVLTree z = y.left;
    x.right = z;
    y.left = x;

    int xb = x.balance;
    int yb = y.balance;
    if (yb < 1)
    { --x.balance;
```

```
        y.balance = ( xb>0 ? yb-1 : xb+yb-2 );
      }
      else if (yb < xb)
      { x.balance -= yb+1;
        --y.balance;
      }
      else y.balance = xb-2;
      return y;
    }
  }
```

例 11.7 构造 AVL 树

图 11.13 表示了向一棵空 AVL 树中插入节点 G、M、T、D 和 P。

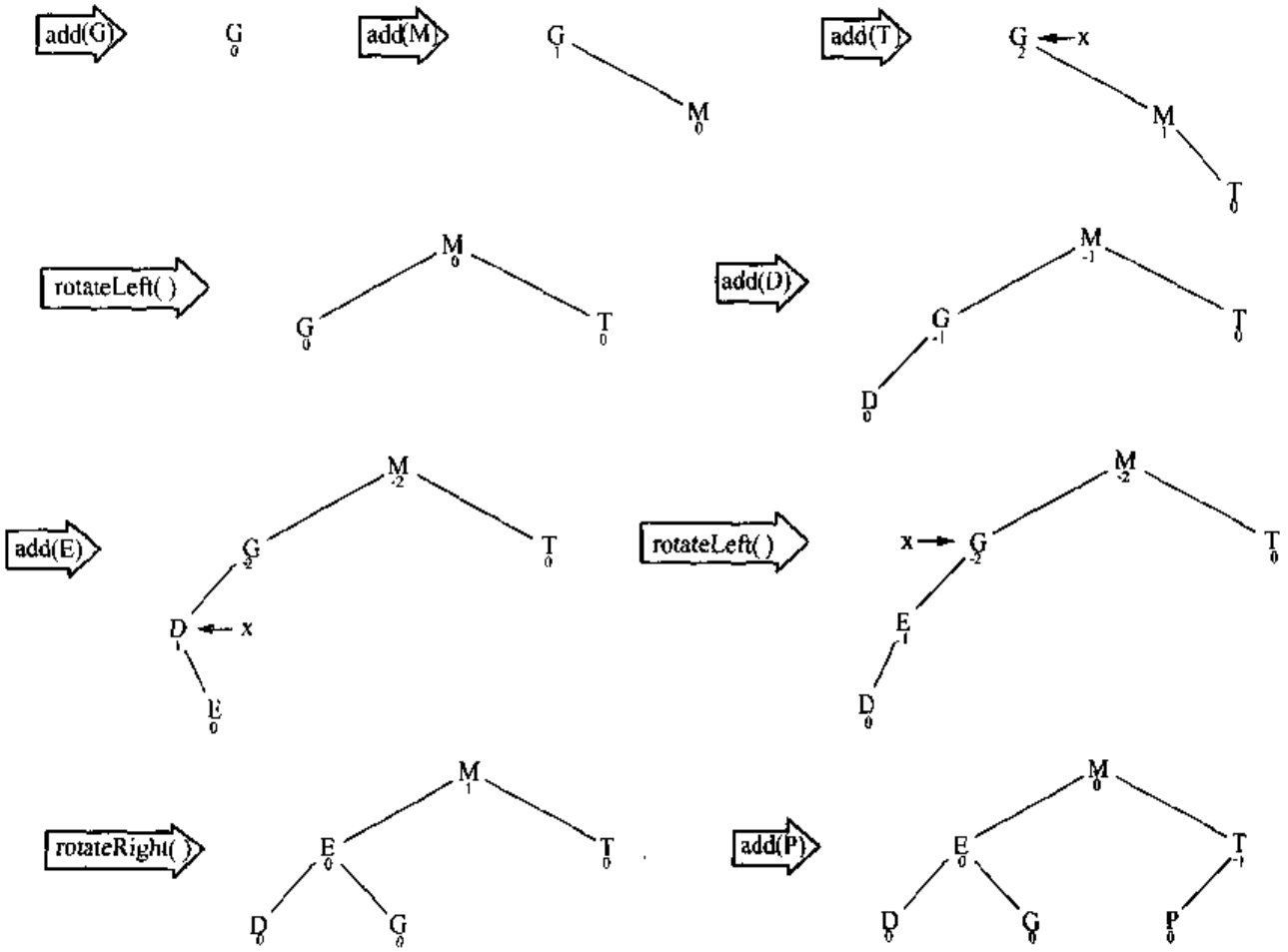


图 11.13 构造 AVL 树

树的第一次旋转发生在插入 T 后。它使得根节点的平衡数增为 2，超过了 AVL 树的限制。在节点 x 出的左旋使得节点 M 成为其父节点 G 的父节点。

第二次旋转发生在 E 插入后。E 的插入导致 G 节点的平衡数为-2，进行一次左旋后得到边 G－E－D，同样有这个问题，还需要再进行一次右旋。类似的情况都需要两次旋转。

标准分享网 www.bzfxw.com 免费下载

旋转算法的执行效率是很高的，只需要对若干对象引用和平衡数的修改就可以使得二叉树近似平衡了。

例 11.8　构造 AVL 树

在图 11.13 的基础上，图 11.14 在插入了 U、V、Z 之后又插入了 W。在这个例子中，除了有两次旋转之外，还有子树转移。包含了 U 的子树从以 V 为父节点转移成以 T 为父节点，以使树的 BST 性得以维持。

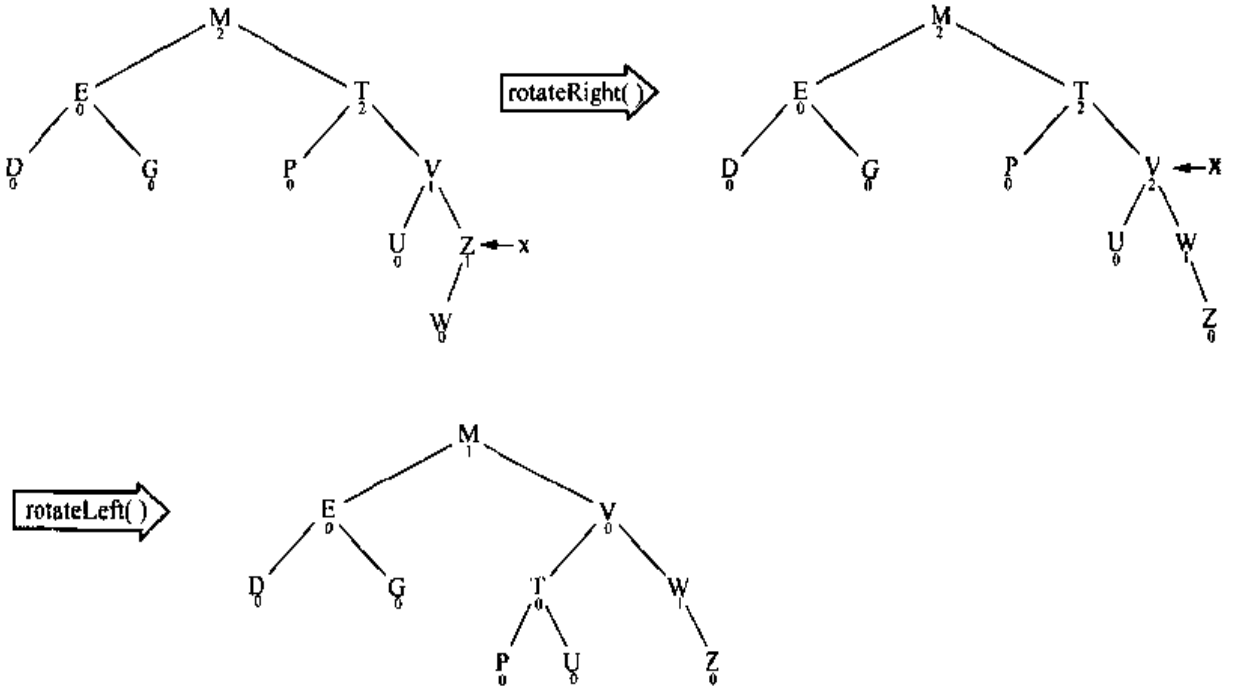


图 11.14　构造 AVL 树

虽然有一些复杂，但是 AVL 树的插入算法是非常高效的。为维持平衡而进行的树的旋转操作也只需要对一些对象引用和平衡数进行修改。

复　习　题

11.1　使用二叉搜索树的优缺点各是什么？

11.2　使用 AVL 树的优缺点各是什么？

习　　题

11.1　如果在例 11.1 的 5 路树中插入一个键值为 16 的记录，情况将会怎样？

11.2　请给出例 11.5 中的 7 个键值的另一组排序，得到与该例子相同的二叉搜索树。

11.3　描述一个基于数组使用二叉搜索树进行排序的算法，并确定算法的复杂性。

11.4　图 11.15 中的二叉树哪些是二叉搜索树？

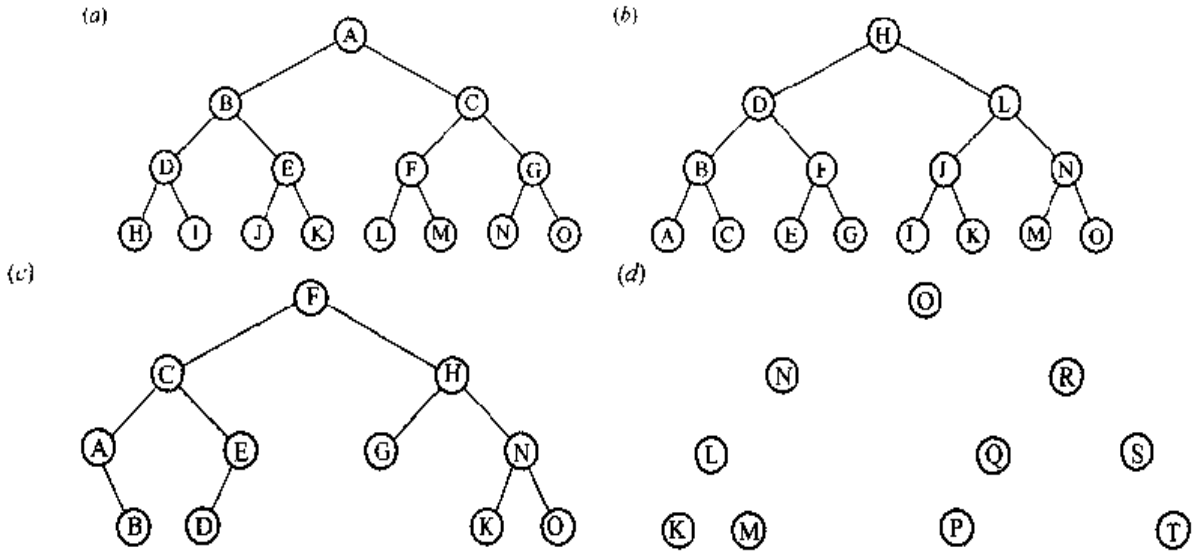


图 11.15 习题 11.4 图

11.5 利用推论 11.2 证明下式：$\log \phi n = 1.44 \lg n$ 。

11.6 为类 AVLTree 编写 `rotateRight()` 方法。

11.7 证明二叉搜索树的任意子树也是二叉搜索树。

11.8 证明 AVL 树的任意子树也是 AVL 树。

11.9 下面是首先认可美国宪法的前 10 个州的邮政缩写：DE，PA，NJ，CA，CT，MA，MD，SC，NH，VA。请给出顺序插入这 10 个缩写所得到的 AVL 树。

复习题答案

11.1 二叉搜索树的缺点是它有可能变得非常不平衡，从而导致搜索时间退化为 $O(n)$的。优点是对于插入和删除操作，二叉搜索树的效率很高。

11.2 AVL 树的优点是它总是平衡的，保证了使用折半查找算法时的 $O(\lg n)$时间复杂性。缺点是在进行插入和删除操作时，需要进行复杂的旋转操作来保证 AVL 树的平衡性。

习 题 解 答

11.1 如果插入一个键值为 16 的记录，开始的搜索过程将定位到第一个叶节点，参照图 11.16（a）。由于这是棵 5 路搜索树，在第一个叶节点上试图进行插入操作将导致它被拆分，并将中间键值 19 上移到父节点中，参照图 11.16（b）。

现在，其父节点也出现覆盖现象，也必须分裂，即将中间键值上移，参照图 11.16（c）。由于是 5 路树，还需要进行两次分裂，最终得到如下形式的树结构，参照图 11.16（d）。

11.2 下面两组排序，都将得到与例 11.5 相同的二叉搜索树。

a. 44，22，33，77，55，99，88

b. 44，22，77，33，55，99，88

标准分享网 www.bzfxw.com 免费下载

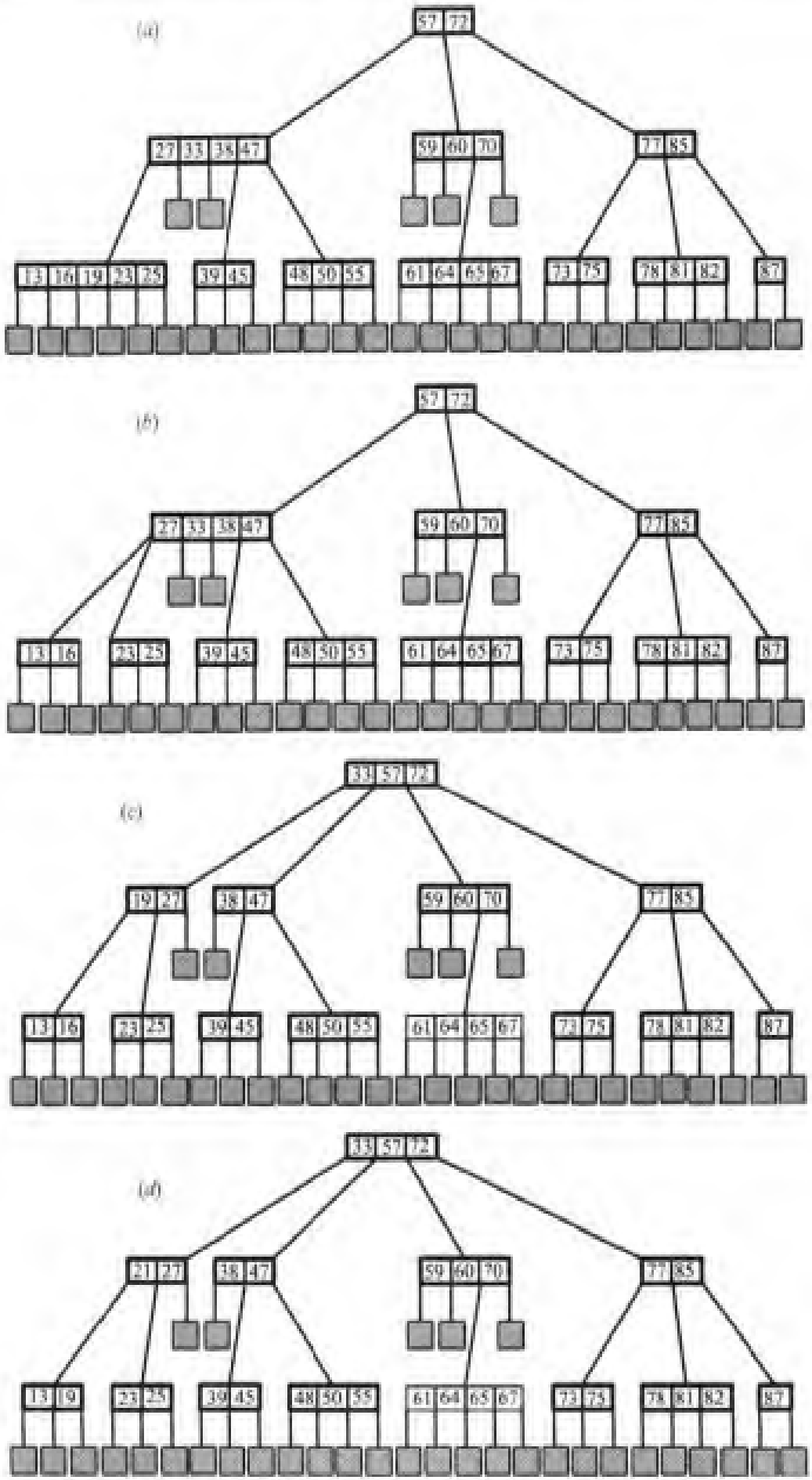


图 11.16 习题 11.1 答案

11.3 一个对象数组可以通过如下方法进行排序：首先将数组中的对象插入一棵二叉搜索树，然后利用中序遍历将数组元素复制回原来的数组中。二叉搜索树的 BST 特性保证了按照中序遍历对树进行遍历时，将按顺序输出元素，从而达到排序目的。

如果我们使用 AVL 树，则每次插入的时间为 $O(\lg n)$，因此建立一棵 n 个元素的 AVL 树将需要 $O(n\lg n)$的时间。中序遍历算法也具有 $O(n\lg n)$时间，因此整个算法对数组的排序将花费 $O(n\lg n)$时间。

11.4 图中的二叉树除了 a 以外，全都是二叉搜索树。

11.5 $\log\ \phi n = (\log \phi 2)(\lg n) = (\ln 2/\ln \phi)(\lg n) = (0.693148/0.481212)(\lg n) = (1.4404)(\lg n) = 1.44\lg n$。

11.6
```
private AVLTree rotateRight()
  { // PRECONDITION: this.left != null;
    BinaryTree x = this;
    BinaryTree y = left;
    BinaryTree z = y.left;
    x.left = z;
    y.left = x;
    if (z != null) z.parent = x;
    y.parent = x.parent;
    x.parent = y;
    int xb = x.balance;
    int yb = y.balance;
    if (yb <= 0)
      if (xb>= 0) y.balance = xb - 2;
      else y.balance = xb - 1;
      { if (xb <= yb) y.balance = xb - 2
        else y.balance = yb - 1;
        x.balance = xb - yb - 1'
      }
      return y;
  }
```

11.7 定理：二叉搜索树的任意子树也是二叉搜索树。

证明：令 T 是二叉搜索树，S 为 T 的子树。令 x 是 S 中的任意元素，L 和 R 分别为 S 中 x 的左、右子树。因为 S 是 T 的子树，x 也是 T 的元素，L 和 R 也分别为 T 中 x 的左、右子树。由于 T 具备 BST 特性，因此，对于每个 $y\in L$，$z\in R$ 有 $y\leqslant x\leqslant z$。同样 S 也具备 BST 特性。

11.8 定理：AVL 树的任意子树也是 AVL 树。

证明：我们在习题 11.7 中已经证明了二叉搜索树的任意子树也是二叉搜索树。假设 S 是 AVL 树 T 的子树，则 S 中的每个节点必然也处在 T 中。因此，S 中的每个节点的平衡数必然是–1、0 或 1。

11.9 解答如图 11.17 所示。

标准分享网 www.bzfxw.com 免费下载

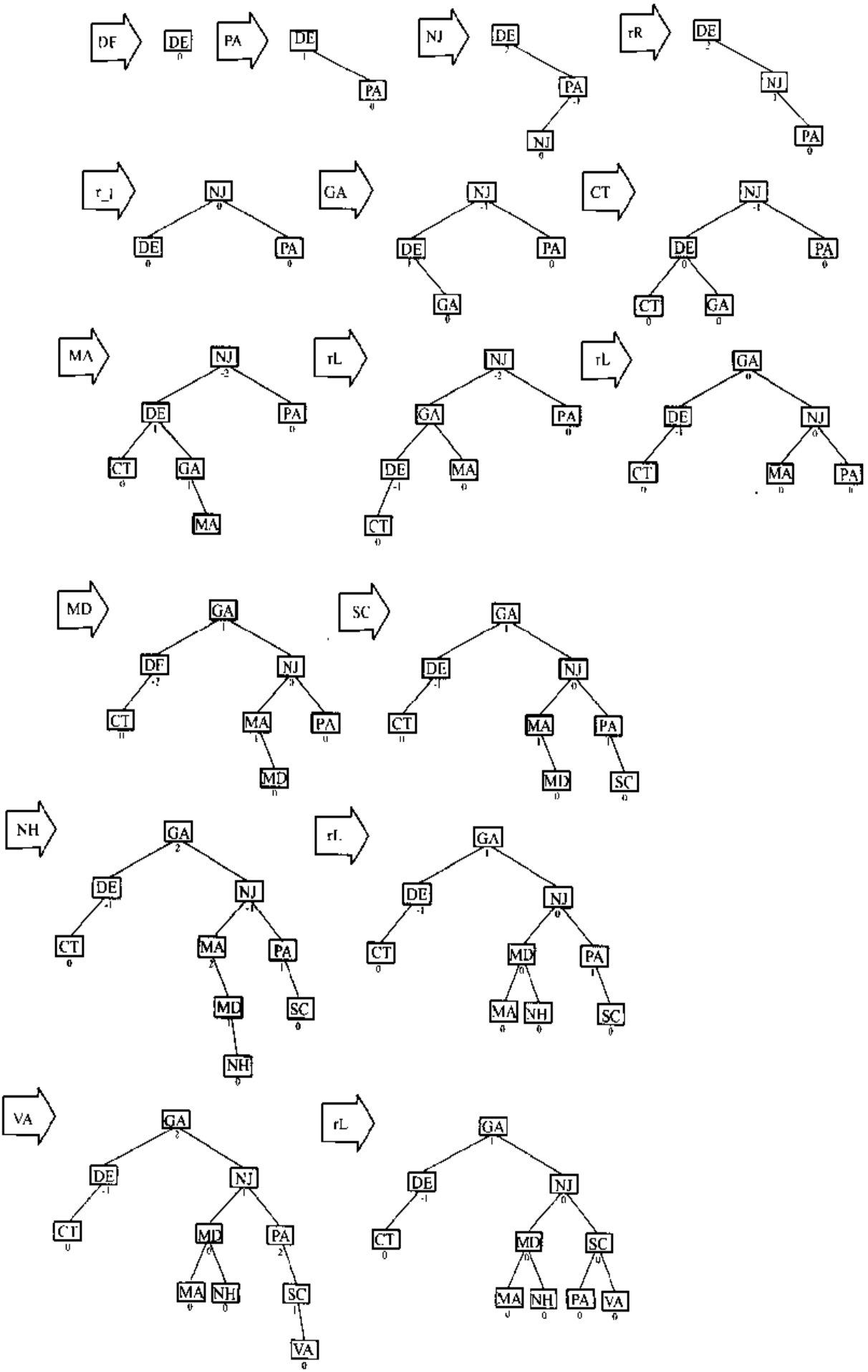


图 11.17　习题 11.9 答案

第 12 章　堆和优先队列

12.1　堆

堆首先是完全二叉树，此外，其元素的键值还满足下列堆性质：从根节点向叶节点的任何路径上的键值是非递增的。例如，图 12.1 给出的二叉树就是一个堆，因为它是一棵完全二叉树，且从根节点到叶节点的任意一条路径上的键值都是非递增的：

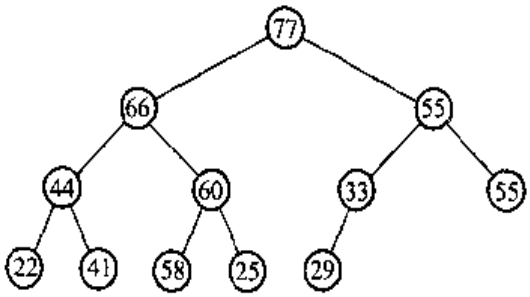


图 12.1　堆

77≥66≥44≥22

77≥66≥44≥41

77≥66≥60≥58

77≥66≥60≥25

77≥55≥33≥29

77≥55≥55

注意到，堆可以用来表示家族树，因为由堆的性质可知每个父节点都比它的孩子节点大。

堆还被用以实现优先队列（参见 12.7 节）和堆算法（参见 13.8 节）。

12.2　自然映射

每个完全二叉树都有一个到数组的自然映射（参见 10.1 节）。比如图 12.1 中的堆映射成如图 12.2 所示的数组。映射实际上是通过对二叉树的一次分层遍历得到的。在所得到的数组中，位于索引 i 处的元素的父节点位于索引 $i/2$ 处，子节点分别位于索引 $2i$ 和 $2i+1$ 处。例如元素 60 位于索引 $i=5$ 处，其父节点位于索引 $i/2=2$ 处，其子节点 58 和 25 分别位于 $2i=10$ 及 $2i+1=11$ 处。

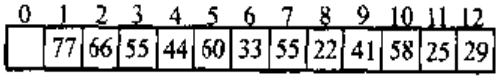


图 12.2　堆到数组的映射

完全二叉树于数组间的这种映射关系是双向的。如果要从数组元素构造对应的完全二叉树，首先将数组第一个元素映射为二叉树的根，然后按照分层遍历策略依次构造出整个二叉树，然后将数组下标 i 中的数据拷贝到第 i 个树节点中即可。如图 12.3 所示。如果所得到的二叉树还具有堆性质，我们称这个数组具有堆性质。

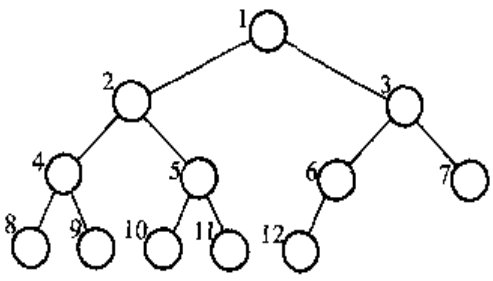


图 12.3　数组到完全二叉树的映射

标准分享网 www.bzfxw.com 免费下载

例 12.1　确定数组是否具有堆性质

为了确定数组是否具备堆性质，我们首先将它转化成二叉树，然后再验证每一条由根到叶的路径，如图 12.4 所示。

路径{88，66，44，51}不符合非递增性，因为 44<51。因此该树不具有堆性质，该数组也就不具有堆性质。

具备堆性质的数组是所谓“部分排序”的。其含义是数组中大部分键值大的元素都位于键值小的元素的前面。更准确地说，其含义是在数组中每个“堆路径子数组”都是非递增排序的，其中，“堆路径子数组”是数组元素的一个子序列，这个子序列中每个元素的下标值都是其下一个元素下标值的一半。例如，{a[1]，a[2]，a[5]，a[11]，a[22]，a[45]，a[90]，a[180]}就是 200 个元素的数组 a[]的一个堆路径子数组。堆排序算法（参见 13.8 节）就是利用这一性质来实现对数组进行快速高效的排序的。

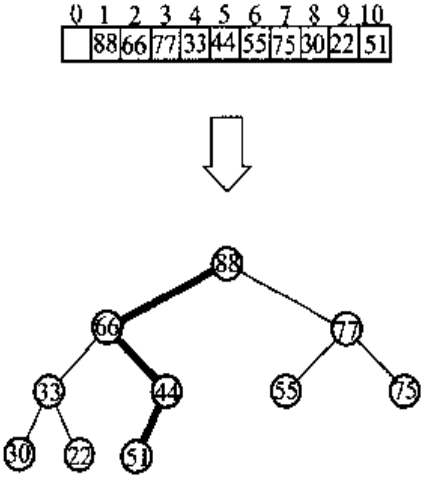


图 12.4　确定数组的堆性质

12.3　堆的插入

在向堆中插入新元素时，新元素总是被插入到最下层，作为最右边的叶节点，然后将这个新元素进行适当的提升使得它不再比它的父节点大。这样，堆性质就得以维护了。每次提升过程是通过与其父节点交换位置完成的。

例 12.2　向堆中插入元素

图 12.5 为向堆中插入键值 75 后的情况。

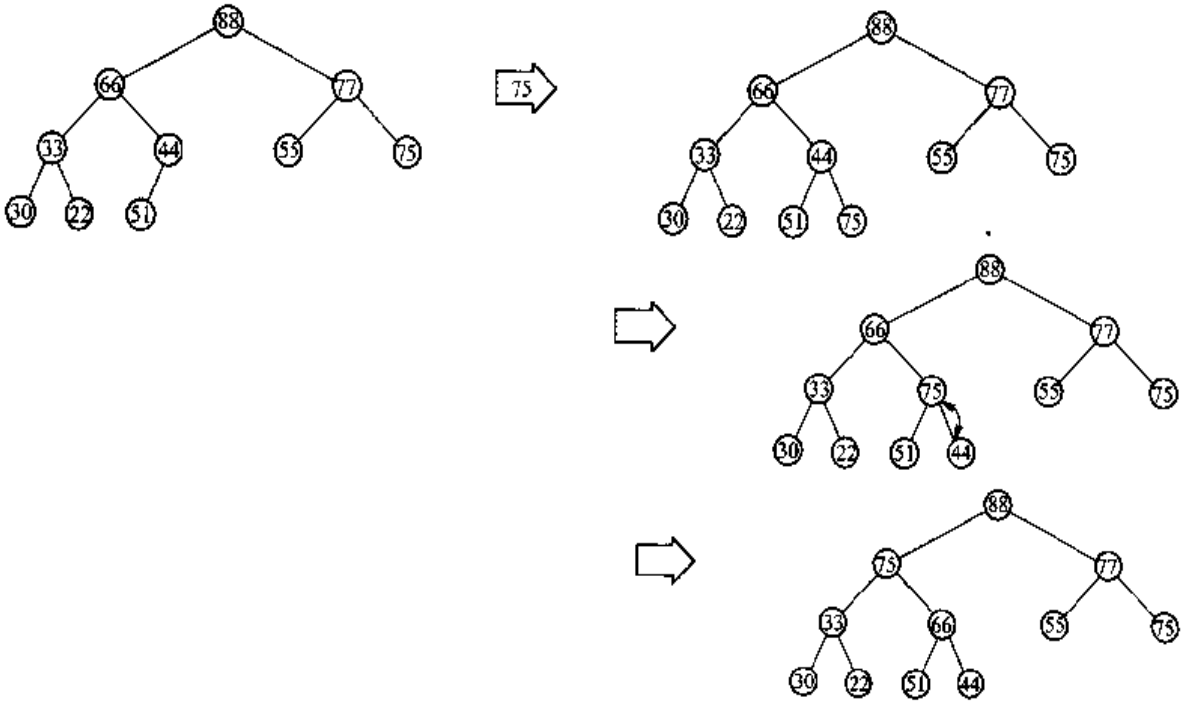


图 12.5　向堆中插入元素

键值 75 是作为最后一个叶节点加入到堆中的。随后，它与父元素 44 交换位置，因为 75>44。然后又与 66 交换位置，因为 75>66。到此为止，堆性质又得以保持，因为 75 现在比父节点小，比子节点大了。

注意到，插入过程只影响到一条由叶到根的路径上的节点。

12.4 堆的删除

堆的删除算法只从树根删除根元素。这一过程是这样进行的：用堆的最后一个叶节点来替代根节点，并将新的根节点不断下降直至它不再比它的子节点小为止，堆的性质又得以保持。每次下降过程是通过与其两个子节点中大者交换位置完成的。

例 12.3 从堆中删除元素

图 12.6 给出的是从堆中删除根元素（键值 88）的过程。

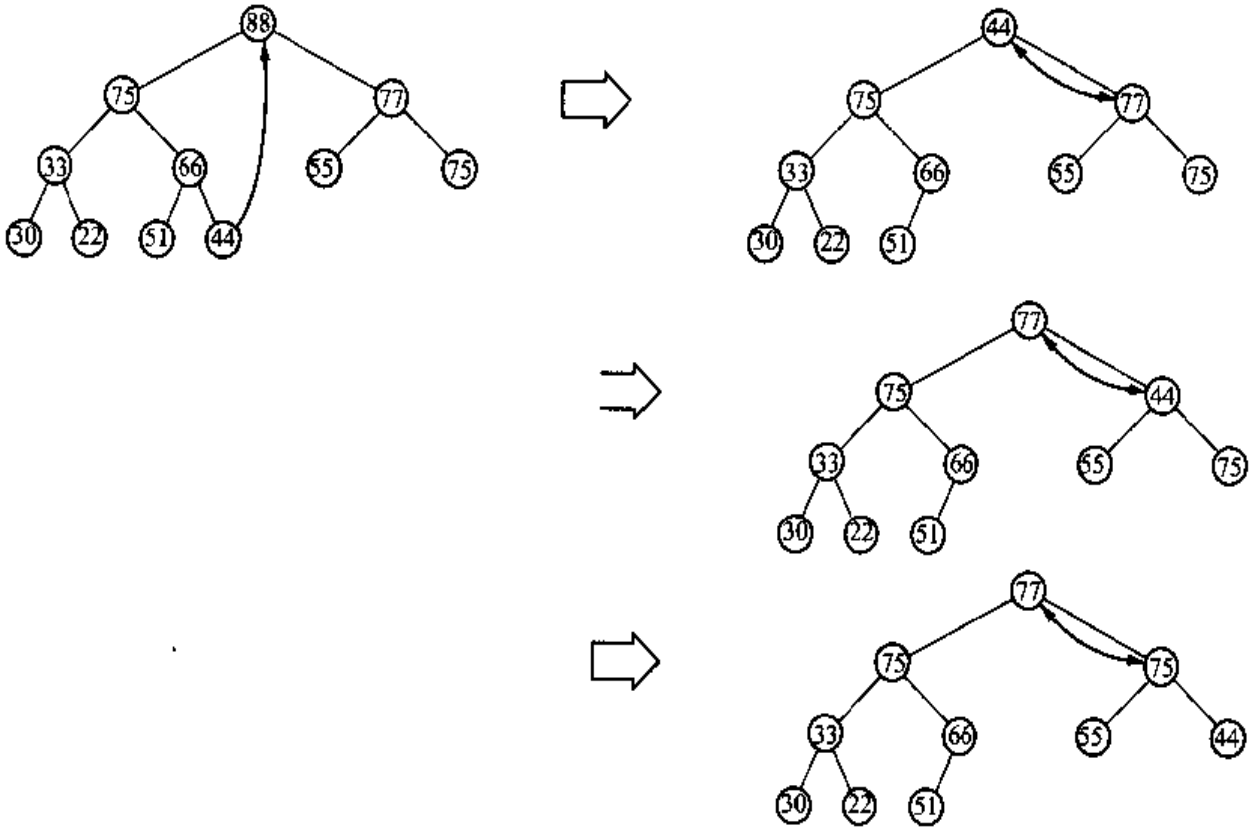


图 12.6 从堆中删除元素

可以看到，堆的最后一个叶节点（键值 44）被用以取代根节点（键值 88）。为了保持堆性质，元素 44 与其当前两个子节点中较大者（77）交换位置。这种过程一直继续到元素 44 不再比它的子节点小为止。在本例中，键值 44 又回到了叶节点的位置。

注意删除过程也只影响到一条由根到叶的路径上的节点。根据定理 10.2 有如下定理：

标准分享网 www.bzfxw.com 免费下载

定理 12.1 向堆中插入元素或从堆中删除元素的时间复杂性是 $O(\lg n)$。

12.5　PriorityQueue 类

堆栈可以被看成一个 LIFO 容器：最后进去的最先出来。队列则是一个 FIFO 容器：最先进去的最先出来。优先队列是一个 BIFO 容器：最有优先权的最先出来。这就是说，每个元素都被赋予一个优先权，具有最高优先权的元素最先出来。

优先队列在计算机系统中的应用非常广。比如，假设同处一个局域网中的若干计算机共享了一台打印机，这台打印机的打印任务一般都暂存在一个优先队列中，而且，短任务的优先权一般比长任务的优先权高。

优先队列一般都用堆来实现，因为堆总是能以键值最大的元素为根节点，而且插入、删除操作都非常高效（参见 12.7 节）。然而，Java Collections Framework 提供了另外一种相当高效的树结构：java.util.TreeSet 类。这使得用 Java 来实现优先队列变得非常简单，其中用到的 Comparator 接口将在下一节介绍。

例 12.4　类 PriorityQueue

下面给出 PriorityQueue 的定义：

```
import java.util.*;

public class PriorityQueue extends TreeSet
{
  public PriorityQueue()
  { super();
  }

  public PriorityQueue(Comparator comparator)
  { super(comparator);
  }

  public void push(Object object)
  { add(object);
  }

  public Object top()
  { return first();
  }
  public Object pop()
  { Object object = first();
```

```
        remove(object);
        return object;
      }
    }
```

作为 TreeSet 的子类，类 PriorityQueue 的插入、删除算法的时间复杂性都是 $O(\lg n)$。

12.6 Java 的 Comparator 接口

优先队列必须能够根据其元素的优先权来对元素进行比较。这一工作在 Java 中已经由 Java Collections Framework 中的 Comparator 对象完成了。

例 12.5 拥有自己的 Comparator 类的类 Person

下面定义的类 Person 描述了人名以及他们的出生年份（yob）、去世年份（yod）。其 toString()方法将按照如下格式打印有关信息：

Pascal (1623 - 1662)

这个类有两个构造函数以防止某些人的去世年份不详（或还未去世）。

```
package schaums.dswj;

pmmmublic class Person
{ String name;
  int yob,yod;
  Person(String name, int yob)
  { this.name = name;
    this.yob = yob;
  }
  Person(String name, int yob, int yod)
  { this(name,yob);
    this.yod = yod;
  }
  public String toString()
  { return name + "(" + yob + (yod>0?"-"+yod:"") + ")";
  }
}
```

下面给出了类 Person 的 Comparator 类。它根据人的出生年份对类 Person 的实例进行比较。

```
Public class YobComparator implements java.util.Comparator
{  public int compare (Object 01,Object 02)
   {  if (01= =null | | !(01 instanceof schaums.dswj. Person)) return -1;
      preson p1 = (Person) 01;
      if (02= =null | | !(02 instanceof Person)) return 1;
```

标准分享网 www.bzfxw.com 免费下载

```
        person p2 = (Person) 02;
        if (p1.getYob () < p2.getYob()) return -1;
        if (p1.getYob () > p2.getYob()) return 1;
        return 0;
      }
}
```

如果第一个人的出生年份早于第二个人，这个类的实例将返回-1，如果晚于第二个人，则返回 1，如果两人同年出生，返回 0。这种返回结果是 Comparator 类与其客户程序间的协议，即负数表示“小于”，“0”表示“等于”，正数表示“大于”。

必须注意的是 Comparator 接口要求按照如下方式声明：

```
public int compare(Object o1, Object o2)
```

因此，这一接口的任何实现都必须检查参数所属的具体类型。这一过程是通过 instanceof 操作完成的。同理，为了能够访问成员变量 yob，参数还必须被强制转换成 Person 类型的对象。

例 12.6　测试带有 Comparator 的类 PriorityQueue

```
import java.util.*;
import schaums.dswj.PriorityQueue;
import schaums.dswj.Person;
import schaums.dswj.YobComparator;

public class Ex1206
{ static public void main(String[] args)
  { PriorityQueue yobQueue = new PriorityQueue(new YobComparator());
    yobQueue.push(new Person("Barrow",1630));
    yobQueue.push(new Person("Fermat",1601));
    yobQueue.push(new Person("Pascal",1623));
    yobQueue.push(new Person("Wallis",1616));
    System.out.println(yobQueue);
    while (!(yobQueue.isEmpty()))
      System.out.println("pop()=" + yobQueue.pop() + ": " + yobQueue);
  }
}
```

上述程序的输出为：

```
[Fermat(1601), Wallis(1616), Pascal(1623), Barrow(1630)]
pop()=Fermat(1601): [Wallis(1616), Pascal(1623), Barrow(1630)]
pop()=Wallis(1616): [Pascal(1623), Barrow(1630)]
pop()=Pascal(1623): [Barrow(1630)]
pop()=Barrow(1630): []
```

上面这段程序首先初始化得到一个 PriorityQueue 对象 yobQueue，并将 YobComparator 类的一个实例与它绑定。带 Comparator 参数的类 TreeSet 的构造函

数负责完成绑定过程。随后，yobQueue 对象通过从 TreeSet 类继承的 add 方法将 4 个 Person 对象加入到优先队列中。

然后，又通过一个循环来测试 yobQueue 的 pop 方法，直到队列为空为止。在每次循环中，通过从 TreeSet 继承来的 toString 方法将队列中的内容打印出来，其中，TreeSet.toString()又调用了 Person.toString()。

只要向其构造函数传递不同的 comparator 对象，同一个类 PriorityQueue 可以与不同的 comparator 绑定在一起使用。

例 12.7 测试带有另一个 Comparator 的类 PriorityQueue

```
import java.util.*;

import schaums.dswj.*;

public class Ex1207
{ static public void main(String[] args)
  { PriorityQueue yodQueue = new PriorityQueue(new YodComparator());
    yodQueue.add(new Person("Barrow",1630,1677));
    yodQueue.add(new Person("Fermat",1601,1665));
    yodQueue.add(new Person("Pascal",1623,1662));
    yodQueue.add(new Person("Wallis",1616,1703));
    System.out.println(yodQueue);
  }
}

class YodComparator implements java.util.Comparator
{ public int compare(Object o1, Object o2)
  { if (o1==null || !(o1 instanceof Person)) return -1;
    Person p1 = (Person)o1;
    if (o2==null || !(o2 instanceof Person)) return  1;
    Person p2 = (Person)o2;
    if (p1.yod<p2.yod) return -1;
    if (p1.yod>p2.yod) return  1;
    return 0;
  }
}
```

上述程序的输出为：

```
[Pascal(1623-1662), Fermat(1601-1665) , Barrow(1630-1677), Wallis(1616-
1703)]
```

PriorityQueue 对象通过父类 TreeSet 来将其中的元素组织成一棵平衡的二叉搜索树。从 TreeSet 中继承来的 TreeSet.toString()方法通过绑定的 comparator，按照增序输出各队列元素的字符串表示。在例 12.6 中，comparator 使用的是 yob，因此 Wallis(1616)

标准分享网 www.bzfxw.com 免费下载

排在 Pascal(1623)之前，而在例 12.7 中，comparator 使用的是 yod，因此 Wallis(1616 - 1703)排在 Pascal(1623 - 1662)之后了。

12.7　优先队列的一个直接实现

12.5 节中实现的优先队列类 PriorityQueue 使用了 Java Collections Framework，因此，这个类的实例存储的元素必须是对象。下面实现了一个面向原始数据类型的优先队列类。

例 12.8　接口 IntPriorityQueueInterface

对于一个整数的优先队列，以下 5 个方法是必须实现的：

```
public interface IntPriorityQueueInterface
{ public int top();
  public int pop();
  public void push(int x);
  public int size();
  public String toString();
}
```

例 12.9　类 IntPriorityQueue

IntPriorityQueue 类通过堆数据结构实现了例 12.8 中的接口：

```
public class IntPriorityQueue implements IntPriorityQueueInterface
{ private final int CAPACITY=16;
  private int capacity, size;
  private int[] ints;

  public IntPriorityQueue()
  { capacity = CAPACITY;
    ints = new int[capacity+1];
  }

  public IntPriorityQueue(int capacity)
  { this.capacity = capacity;
    ints = new int[capacity+1];
  }

  public int top()
  { return ints[1];
  }

  public int pop()
  { int x = ints[1];
```

```
    ints[1] = ints[size--];
    heapifyDown();
    return x;
  }

  public void push(int x)
  { if (size==capacity) resize();
    ints[++size] = x;
    heapifyUp();
  }

  public int size()
  { return size;
  }

  public String toString()
  { if (size==0) return "[]";

    String string = "[" + ints[1];
    for (int i=2; i<=size; i++)
      string += "," + ints[i];
    return string + "]";
  }

  private void heapifyUp()
  { for (int j=size, i; j>0; j = i)
    { i = j/2;
      if (ints[j]<ints[i]) swap(ints,i,j);
    }
  }
    private void heapifyDown()
  { for (int i=1, j; i<=size/2; i = j)
    { j = 2*i;
      if (j<size && ints[j]>ints[j+1]) ++j;
      if (ints[j]<ints[i]) swap(ints,i,j);
    }
  }

  private void resize()
  { int[] temp = new int[capacity+1];
    for (int i=1; i<=capacity; i++)
      temp[i] = ints[i];
    capacity *= 2;
    ints = new int[capacity+1];
```

标准分享网 www.bzfxw.com 免费下载

```
    for (int i=1; i<=size; i++)
      ints[i] = temp[i];
  }

  private void swap(int[] a, int i, int j)
  { int tmp=a[i];
    a[i] = a[j];
    a[j] = tmp;
  }
}
```

这段代码通过数组将优先队列实现为堆数据结构。size 域存放的是优先队列中已存放元素的个数。元素被存放在数组 ints[1..size]中，也就是说，本实现使用了从 1 开始的数组下标编码方式（而不是从 0 开始的数组下标编码方式）来简化堆数据结构的各种操作，有关内容参见第 12.2 节算法 10.1。

为了让数组能够有效地增长，在实现中我们使用了所谓“数组容量加倍”的技巧。这一技巧是通过引入一个单独的容量变量即 *capacity* 实现的。具体来说就是 capacity 的作用是为了确定数组 ints[]的实际大小。前面已经讲过，我们使用的是从 1 开始的数组下标编码方式，因此，数组的实际大小应该是 capacity+1。capacity 的被初始化成常量 CAPACITY（在本例中 CAPACITY 为 16，也可以是其他正整数）。所以，ints[]数组的初始实际大小是 17。这时的优先队列最多可以容纳 16 个元素。在 size= =capacity 成立的条件下调用 push()操作时，私有方法 resize()将优先队列的容量增加了一倍。

push()操作的实现体现了第 12.3 节中算法的设计思想，而 pop()操作的实现体现了第 12.4 节中算法的设计思想。它们分别用到了私有方法 heapifyUp()和 heapifyDown()。

例 12.10　测试类 IntPriorityQueue

下面的程序段对例 12.9 中的类 IntPriorityQueue 进行测试：

```
public class Ex1210
{ public static void main(String[] args)
  { IntPriorityQueue queue = new IntPriorityQueue();
    queue.push(50);
    queue.push(95);
    queue.push(70);
    queue.push(30);
    queue.push(90);
    queue.push(25);
    queue.push(55);
    queue.push(35);
    queue.push(80);
    queue.push(60);
    queue.push(40);
```

```
        queue.push(20);
        queue.push(10);
        queue.push(75);
        queue.push(45);
        queue.push(65);
        System.out.println(queue.size() + ": " + queue);
        queue.push(85);
        System.out.println(queue.size() + ": " + queue);
        queue.pop();
        System.out.println(queue.size() + ": " + queue);
        while(queue.size()>0)
          System.out.println("queue.pop() = " + queue.pop());
        System.out.println(queue.size() + ": " + queue);
      }
    }
```

上述程序的输出为：

```
16:[10,35,20,50,40,25,45,65,80,90,60,70,30,75,55,95]
17:[10,35,20,50,40,25,45,65,80,90,60,70,30,75,55,95,85]
16:[20,35,25,50,40,30,45,65,80,90,60,70,85,75,55,95]
queue.pop() = 20
queue.pop() = 25
queue.pop() = 30
queue.pop() = 35
queue.pop() = 40
queue.pop() = 45
queue.pop() = 50
queue.pop() = 55
queue.pop() = 60
queue.pop() = 65
queue.pop() = 70
queue.pop() = 75
queue.pop() = 80
queue.pop() = 85
queue.pop() = 90
queue.pop() = 95
0: []
```

在将变量queue初始化成IntPriorityQueue对象之后，程序调用其push()方法向优先队列中加入了17个元素。println()方法又激活了IntPriorityQueue的toString()方法，后者依次将数组ints[]中的元素打印出来。例如，程序输出的第一行：

```
16:[10,35,20,50,40,25,45,65,80,90,60,70,30,75,55,95]
```

说明队列中有16个元素并随后一一列举出来。从这个例子中我们还可以来检查一下数

标准分享网 www.bzfxw.com 免费下载

组的堆性质是否成立。例如，从根到叶的路径 ints[1,2,4,8,16]对应的递增序列是[10, 35, 50, 65, 95]，另外一条路径 ints[1,3,6,13]对应的递增序列是[10,20,25,70]。（为了更全面的了解每条路径，可参见问题 12.7）

第二行输出说明 resize()方法达到了预期目的。最后一条 push 语句即 push(85)在数组已满的情况下（size == capacity）试图插入元素。因此，push 方法调用 resize()方法在元素插入之前将数组 ints[]的容量增加了一倍。

接下来的 pop()操作将从优先队列中删除一个元素。这一删除操作删除的是具有最高优先权的元素（最小的整数），即整数 10。这是因为这个优先队列是用堆数据结构实现的，最小的元素处于根节点的位置，即数组 ints[1]的位置。元素的删除算法（参见第 12.4 节）将数组的最后一个元素移到的根节点位置，然后调用 heapifyDown()方法来维护数组的堆性质。这一过程的结果是使得元素 85 又落回到了叶节点的位置，同时使得路径 ints[1,3,6,13]上的序列由[85,20,25,30]变成[20,25,30,85]。

最后，通过一个 while 循环将优先队列中的所有元素一一弹出，其中共 16 次调用了 pop()操作。请注意，元素是按照增序被弹出的：20,25,30,35,...,95。

例 12.10 的输出给我们一个快速排序的方法：将一组数据插入优先队列，再弹出它们即可。假设优先队列是通过堆数据结构实现的，这 pop()操作的时间复杂性是 $O(\lg n)$。执行 n 次的时间复杂性则为 $O(n \lg n)$。n 次插入操作的时间复杂性为 $O(n)$，因此，整个排序的时间复杂性是 $O(n \lg n)$。这个排序算法称为堆排序。（参见第 13.8 节）

复　习　题

12.1　堆的两个主要应用是什么？

12.2　向堆中插入元素、从堆中删除元素的算法效率如何？

12.3　优先队列为什么被称为 BIFO 容器？

12.4　队列与优先队列间的差别在哪里？

12.5　为什么用堆来实现优先队列？

12.6　在二叉树到数组 a[]的自然映射中，为什么从 a[1]开始存储而不从 a[0]开始存储？

12.7　在类 IntPriorityQueue 的实现中，当 push()方法被调用且数组已满时，将通过“数组容量加倍”技巧来将数组容量增加一倍。类似地，在 pop()操作被调用且数组实际大小只有容量的一半时，也可以通过“数组容量减半”技巧来将数组容量减少一半。试分析这样做有没有意义？

习　　题

12.1　试确定图 12.7 中的二叉树哪些是堆？

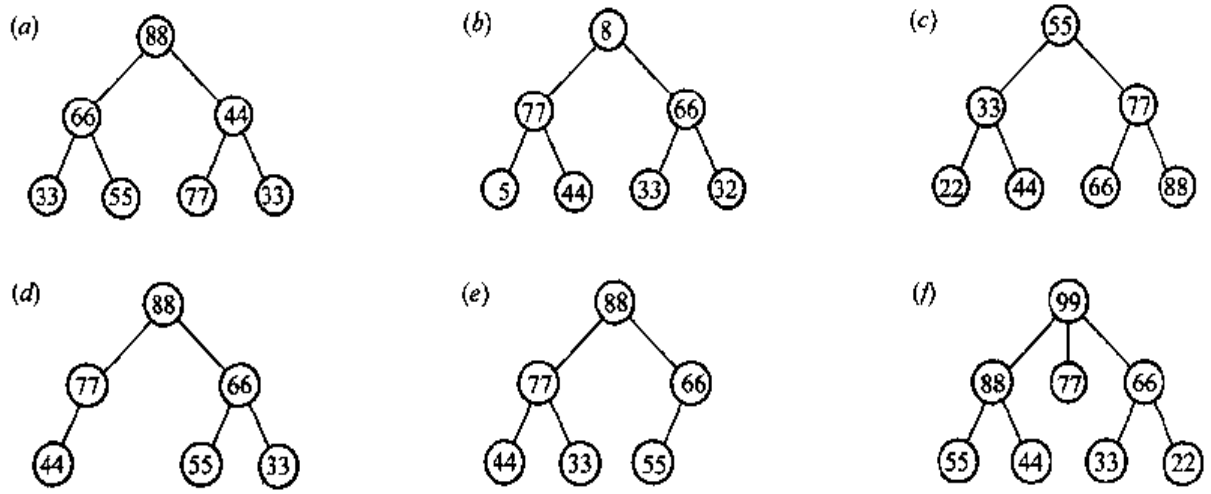


图 12.7 习题 12.1 图

12.2 试确定图 12.8 中的数组哪些具有堆性质?

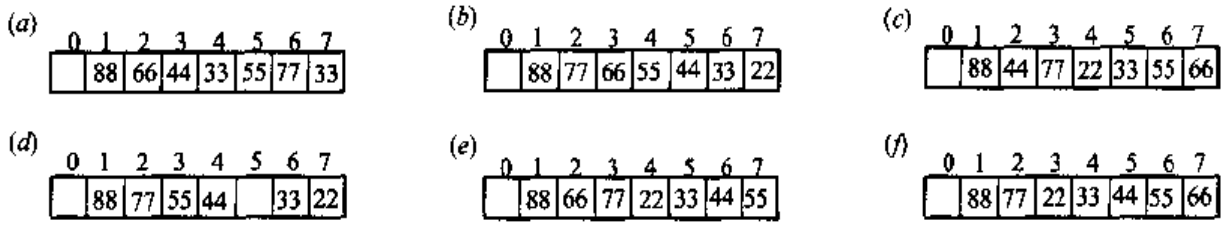


图 12.8 习题 12.2 图

12.3 按照顺序画出依次插入元素 44，66，33，88，77，55，22 的堆的结构。

12.4 分别画出问题 12.3 中各堆在自然映射下对应的数组的结构。

12.5 试证明堆的各子树都是堆。

12.6 为例 12.9 中定义的类 IntPriorityQueue 加入并测试如下方法:

```
public boolean isHeap()
// 当且仅当数组 ints[]满足堆性质，返回 true
```

12.7 依照例 12.10 的执行程序线索，画出每一步对应的堆及数组表示。

12.8 修改例 7.7 的客户/服务器的模拟程序，使得它使用的是优先队列而不是一般队列。为 Client 类定义 Comparator 类以使短作业有较高的优先级。

复习题答案

12.1 堆常用于实现优先队列和堆排序算法（参见算法 13.8）。

12.2 向堆中插入元素、从堆中删除元素的算法是非常高效的，只需 $O(\lg n)$的时间。

12.3 优先队列被称为 BIFO 容器，因为队列中具有最高优先权的元素首先从队列中被取出。

标准分享网 www.bzfxw.com 免费下载

12.4　队列中的元素在从队列中被取出时，是按照它们被插入的顺序的，也就是说是先进先出的。优先队列中的元素在从优先队列中被取出时，是按照它们的优先权的顺序进行的。

12.5　用堆来实现优先队列是因为堆的插入和删除只需 $O(\lg n)$的时间。这是因为 push()和 pop()方法的实现都是遍历了堆中一条由根到叶的路径，这条路径最多不过堆的高度即 $\lg n$。

12.6　在二叉树到数组 a[]的自然映射中，从 a[1]开始存储而不从 a[0]开始存储是为了方便对堆中元素的访问。通过为根节点赋值为 1 并依次排序，任意节点 k 的父节点的值将为 $k/2$，子节点的值分别为 $2k$ 和 $2k+1$。

12.7　在类 IntPriorityQueue 的实现中，当 push()方法被调用且数组已满时，将通过“数组容量加倍”技巧来将数组容量增加一倍是必要的。但是在 pop()操作被调用且数组实际大小只有容量的一半时，通过“数组容量减半”技巧来将数组容量减少一半却没有必要，因为这样做虽然能够减少一些存储空间，但是算法带来的额外开销却很大。

习 题 解 答

12.1　确定下面的二叉树哪些是堆。

a. 不是堆，因为从根到叶的路径{88，44，77}不是非递增的（44<77）。

b. 是堆。

c. 不是堆，因为从根到叶的路径{55，33，44}不是非递增的（33<44），而且从根到叶的路径{55，77，88}不是非递增的（55<77<88）。

d. 不是堆，因为该二叉树不完全。

e. 是堆。

f. 不是堆，因为该树不是二叉树。

12.2　确定下面的数组哪些具有堆性质。

a. 不具有堆性质，因为从根到叶的路径{a[1]，a[3]，a[6]}={88，44，77}不是非递增的（44<77）。

b. 具有堆性质。

c. 具有堆性质。

d. 不具有堆性质，因为数组的元素不是连续的：它不能表示完全二叉树。

e. 具有堆性质。

f. 不具有堆性质，因为从根到叶的路径{a[1]，a[3]，a[6]}={88，22，55}不是非递增的（22<55），从根到叶的路径{a[1]，a[3]，a[7]}={88，22，66}不是非递增的（22<66）。

12.3　依次插入元素 44，66，33，88，77，55，22 的堆的结构如图 12.9 所示。

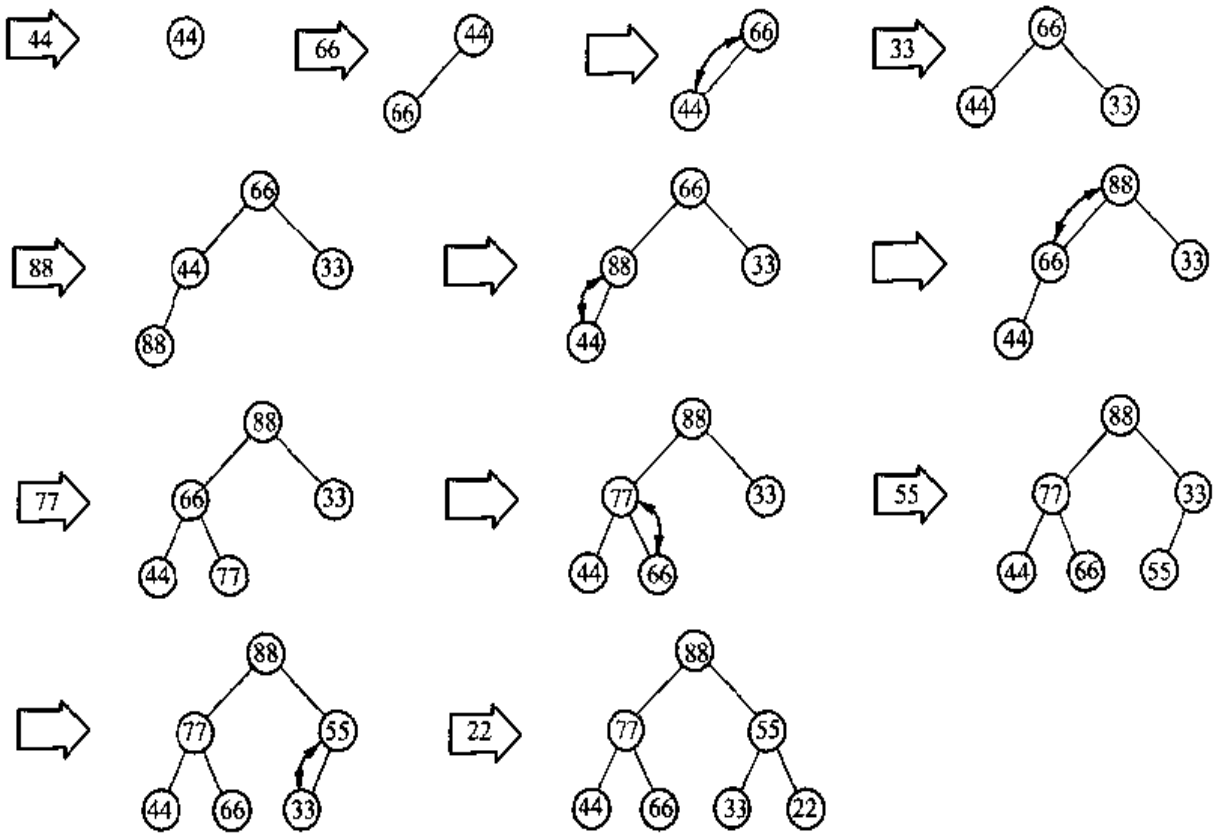


图 12.9 习题 12.3 答案

12.4 习题 12.3 中各堆在自然映射下对应的数组的结构如图 12.10 所示。

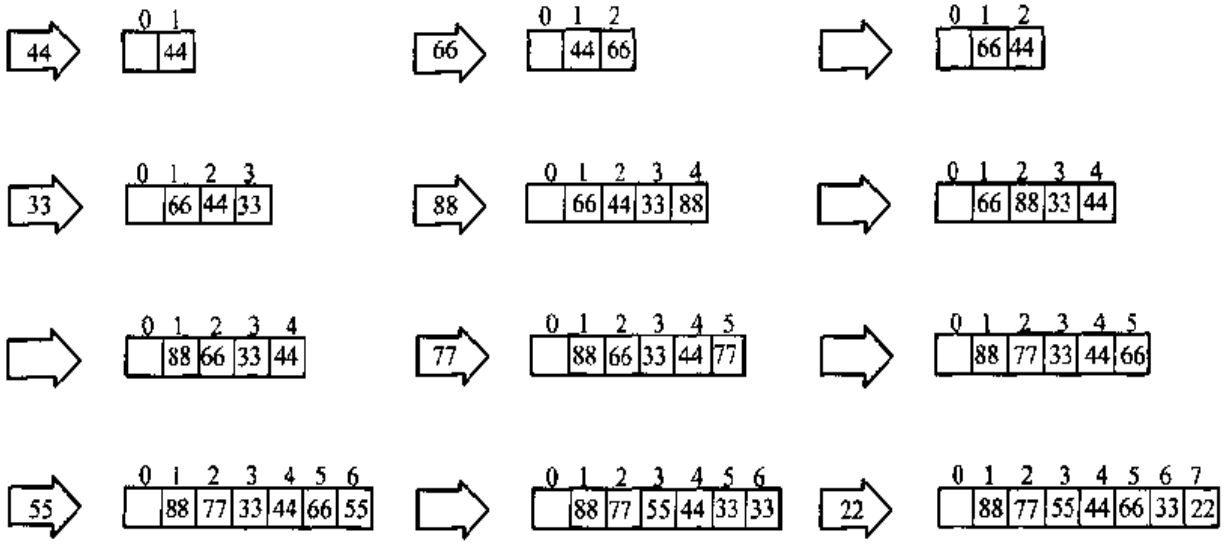


图 12.10 习题 12.4 答案

12.5 定理：堆的各子树都是堆。

证明： 令 T 是堆，S 为 T 的子树。图 12.11 所示。根据定义，T 是具有堆性质的完全二叉树。根据第 10 章习题解答中的结论，S 也是完全二叉树。令 x 是 S 的根元素，令 p 为 S 中任意由根到叶的路径，则因为 S 是 T 的子树 x 也是 T 的元素，并且在 T 中存在惟一路径 q 从 x 到 T 的根节点。由于 S 是 T 的子树，

标准分享网 www.bzfxw.com 免费下载

p 也是 T 中的路径，它连接了 x 到 T 的某个叶节点。令 q^{-1} 表示路径 p 的反向路径，令 $q^{-1}p$ 表示路径 q^{-1} 与 p 的连接。则路径 $q^{-1}p$ 是 T 中从根到叶的路径。因此，路径 $q^{-1}p$ 上的元素必然是非递增的，因为 T 具有堆性质。因而，路径 p 也是非递增的，所以 S 也具备堆性质。

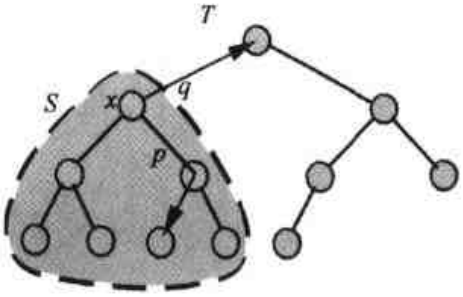


图 12.11　堆与其子树

12.6　程序如下：

```
public boolean isHeap()
{  for (int leaf=size/2+1; leaf≤size; leaf++)
   for(int j=leaf, i; j>0; j = i)
   {  i = j/2;
      if (ints[j]<ints[i])  return false;
   }
   return true;
}
```

12.7　图 12.12 给出了例 12.10 的执行程序线索。

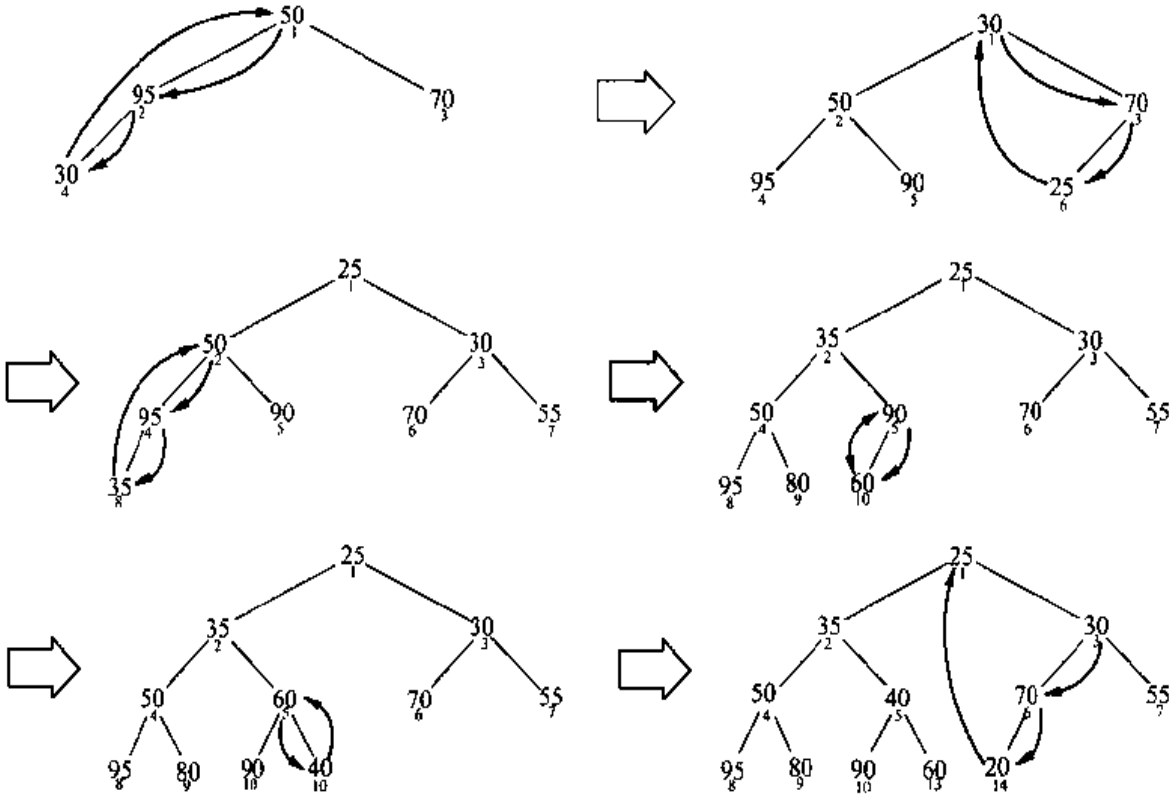


图 12.12　习题 12.7 答案

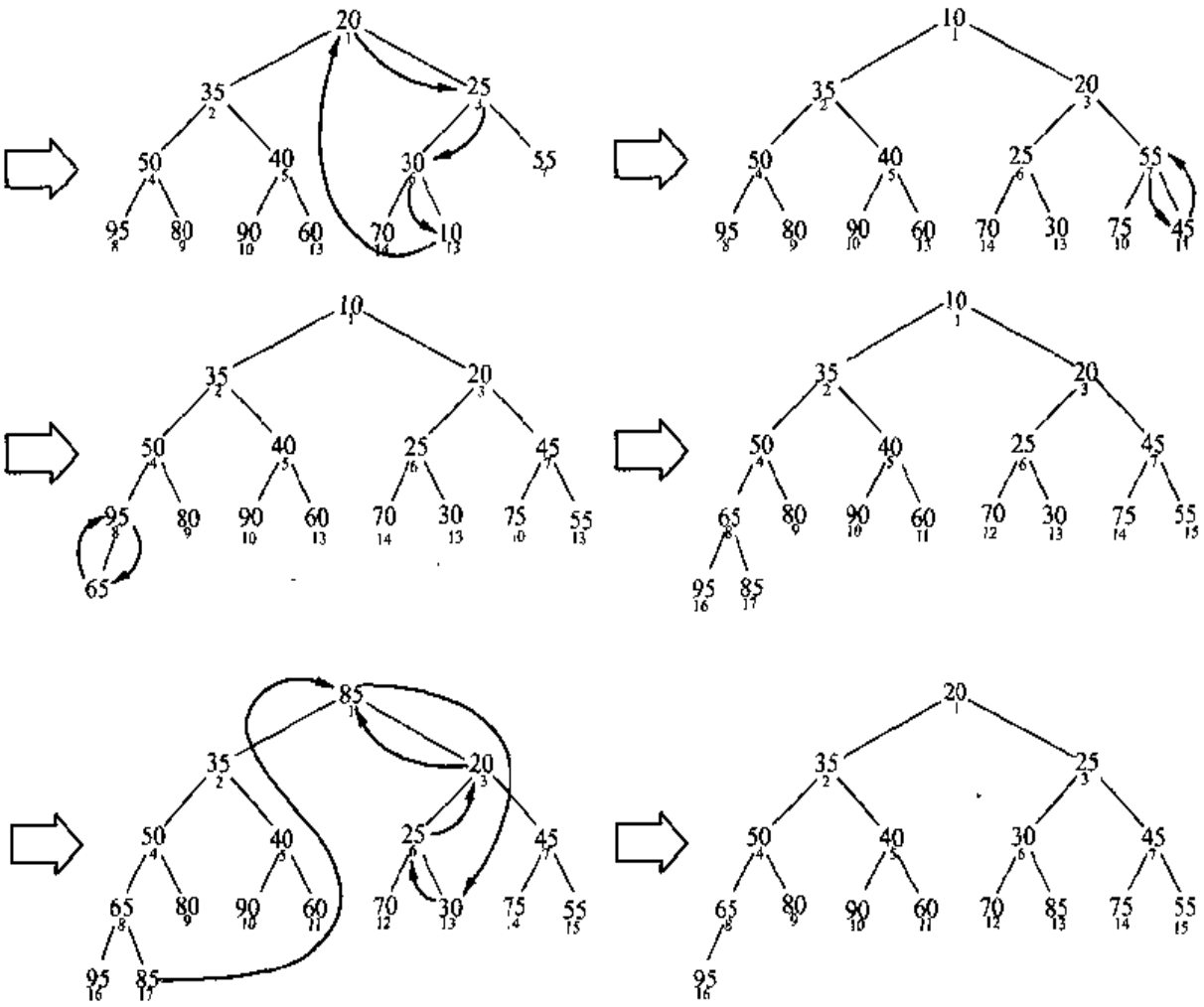


图 12.12 习题 12.7 答案（续）

12.8 程序如下：

```
import schaums.dswj.*;
import schaums.dswj.PriorityQueue;

public class Pr1208
{ private static final int NUMBER_OF_SERVERS = 4;
  private static final double MEAN_INTERARRIVAL_TIME = 5.0;
  private static final int DURATION = 200;
  private static Server[] servers = new Server[NUMBER_OF_SERVERS];
  private static PriorityQueue clients
     = new PriorityQueue(new ClientComparator());
  private static Random random
     = new Random(MEAN_INTERARRIVAL_TIME);

  public static void main(String[] args)
  { for (int i=0; i<NUMBER_OF_SERVERS; i++)
        servers[i] = new Server();
    int timeOfNextArrival = random.intNextExponential();
```

标准分享网 www.bzfxw.com 免费下载

```
        for (int t=0; t<DURATION; t++)
        { if (t == timeOfNextArrival)  // new arrival
            { clients.push(new Client(t));
              timeOfNextArrival += random.intNextExponential();
              System.out.println(t + "\t" + clients);
            }
            for (int i=0; i<NUMBER_OF_SERVERS; i++)  // process servers
              if (servers[i].isFree())
              { if(!clients.isEmpty())
                    servers[i].beginServing((Client)clients.pop(),t);
              }
              else if (t == servers[i].getTimeServiceEnds())
                servers[i].endServing(t);
        }
    }
}

public class Server
{ private static Random randomMeanServiceRate
    = new Random(1.00,0.20);
  private static char nextId = 'A';
  private Random randomServiceRate;
  private char id;
  private double meanServiceRate, serviceRate;
  private Client client;
  private int timeServiceEnds;

  public Server()
  { id = (char)nextId++;
    meanServiceRate = randomMeanServiceRate.nextGaussian();
    randomServiceRate = new Random(meanServiceRate,0.10);
  }

  public Client getClient()
  { return client;
  }

  public void beginServing(Client client, int time)
  { this.client = client;
    serviceRate = randomServiceRate.nextGaussian();
    client.beginService(this,time);
    int serviceTime
      = (int)Math.ceil(client.getJobSize()/serviceRate);
    timeServiceEnds = time + serviceTime;
```

```
    System.out.println(time + "\t" + id + " starts serving "
                       + client);
  }

  public void endServing(int time)
  { client.endService(time);
    System.out.println(time + "\t" + id + " finishes serving "

                       + client);
    this.client = null;
  }

  public int getTimeServiceEnds()
  { return timeServiceEnds;
  }

  public boolean isFree()
  { return client == null;
  }

  public String toString()
  { int percentMeanServiceRate
        = (int)Math.round(100*meanServiceRate);
    int percentServiceRate = (int)Math.round(100*serviceRate);
    return id + "(" + percentMeanServiceRate + "%,"
           + percentServiceRate + "%)";
  }
}

public class ClientComparator implements java.util.Comparator
{ public int compare(Object o1, Object o2)
  { if (o1==null || !(o1 instanceof schaums.dswj.Client))
      return -1;
    Client c1 = (Client)o1;
    if (o2==null || !(o2 instanceof Client)) return  1;
    Client c2 = (Client)o2;
    if (c1.getJobSize() < c2.getJobSize()) return -1;
    if (c1.getJobSize() > c2.getJobSize()) return  1;
    return 0;
  }
}

public class Client
{ public static final int MEAN_JOB_SIZE = 20;
```

标准分享网 www.bzfxw.com 免费下载

```
private static Random randomJobSize = new Random(MEAN_JOB_SIZE);
private static int nextId = 0;
private int id, jobSize, tArrived, tBegan, tEnded;
private Server server;

public Client(int time)
{ id = ++nextId;
  jobSize = randomJobSize.intNextExponential();
  tArrived = time;
  System.out.println(time + "\t" + this + " arrives");
}

public double getJobSize()
{ return jobSize;
}
public int getWaitTime()
{ return tBegan - tArrived;
}

public int getServiceTime()
{ return tEnded - tBegan;
}

public void beginService(Server server, int time)
{ this.server = server;
  tBegan = time;
}

public void endService(int time)
{ tEnded = time;
  server = null;
}

public String toString()
{ return "#" + id + "(" + (int)Math.round(jobSize) + ")";
}

private static void print(int job, int time, double size)
{ System.out.println("Job #" + job + " arrives at time " + time
    + " with " + (int)Math.round(size) + " pages.");
}

private static void printBegins(Server server, int job, int time)
{ System.out.println("Printer " + server + " begins Job #" + job
```

```
        + " at time " + time + ".");
    }

    private static void printEnds(Server server, int job, int time)
    { System.out.println("Printer " + server + " ends Job #" + job
        + " at time " + time + ".");
    }
}
```

标准分享网 www.bzfxw.com 免费下载

第13章 排　序

从第 2 章的讨论中可以看出，当一个数组中元素已被排序后，在这个数组中作搜索是非常高效的。这就好像是在一个电话号码簿中查找某个电话号码，或者从一本字典中查找某个单词一样。在本章中，我们将讨论 10 种运用得最为广泛的线性结构（比如数组或者列表）的排序算法。

本章中介绍的所有排序算法都是采用下面这个 Java 方法声明方式来对一个整数数组进行排序的：

```
public static void sort (int[ ] a)
```

这可以通过简单的修改来实现对其他基本类型的数组进行排序，或对实现了 Comparable 接口的对象数组进行排序。对列表及其他数据结构进行排序的内容可以参见后续的有关例子。

为了更好地证明算法的正确性以及分析算法的复杂性，本章的伪码和 Java 代码都包含了关于前置条件、后置条件和循环不变量的说明。

例如，交换排序就利用了下面这个通用方法：

```
private static void swap(int[ ] a, int i, int j)
{
//PRECONDITIONS: 0 <= i < a.length; 0 <= j < a.length;
//POSTCONDITION: a[i] and a[j] are interchanged;
if (i == j) return;
int temp = a[j];
a[j] = a[i]j;
a[i] = temp;
}
```

说明这个方法仅仅将数组的第 i 个元素与第 j 个元素作一个简单的交换。

在伪码中，我们用记法 $s=\{s_0, s_1, ..., s_{n-1}\}$ 来表示一个 n 个元素的序列，记法 $\{s_p..s_q\}$ 来表示从元素 s_p 到元素 s_q 的子序列 $\{s_p, s_{p+1}, ..., s_q\}$。在 Java 代码的注解中，我们用 s[p..q] 来表示这样的一个子序列。例如记法 $\{s_3..s_7\}$ 和 s[3..7] 都表示子序列 $\{s_3, s_4, s_5, s_6, s_7\}$。

除非特别声明，“排序”都是指将序列的元素按照非递减序排列，即 $s_0 \leqslant s_1 \leqslant \cdots \leqslant s_{n-1}$。

13.1 Java 中的 Arrays.sort()方法

在标准的 Java 类库中，类 java.util.Arrays 中定义了 sort()方法。这一方法实

际上有 18 个覆盖版本，其中 4 个是为对象数组定义的，另外，除了 boolean 类型以外的其他基本类型都定义了两个方法。例如，对于 int 类型的两个方法的声明如下：

```
public static void sort(int[] a)
public static void sort(int[] a, int p, int q)
```

其中，第一个方法对整个数组排序，第二个方法对子数组 a[p..q]排序。

例 13.1 Arrays.sort()方法的使用

```
public class Ex1301
{ public static void main(String[] args)
  { int[] a = { 77, 44, 99, 66, 33, 55, 88, 22 };
    print(a);
    java.util.Arrays.sort(a);
    print(a);
  }

  private static void print(int[] a)
  { for (int i=0; i<a.length; i++)
      System.out.print(a[i]+" ");
    System.out.println();
  }
}
```

上述程序的输出为：

```
77 44 99 66 33 55 88 22
22 33 44 55 66 77 88 99
```

Arrays.sort()方法实现的是快速排序（参见算法 13.6）。

13.2 冒泡排序

冒泡排序对 n 个元素的序列进行 $n-1$ 次扫描。在每次扫描中，它都从左到右对相邻的两个元素进行比较，如果它们不符合顺序，就将它们交换位置。这一过程将使得大的元素渐渐地移动到序列的最右边。这一方法之所以被称为“冒泡排序”，是因为如果我们将这个序列垂直放置，那么每次扫描过程就好像是汽水中气泡在不断向上冒一样：大元素总是不断与相邻的小元素交换位置，向上“浮动”并最终达到它应该在的位置（参见图 13.1）。



图 13.1 冒泡

算法 13.1 冒泡排序

（前置条件：$s=\{s_0, s_1, s_2, ..., s_{n-1}\}$是 n 个可排序元素的序列）

标准分享网 www.bzfxw.com 免费下载

（后置条件：序列 s 被排序）

1. 令 i 从 n-1 递减到 1，重复步骤 2~4。
2. 令 j 从 1 递增到 i，重复步骤 3。
3. 如果元素 s_{j-1} 与 s_j 呈反序，交换它们。
4. （不变量：子序列$\{s_i, ..., s_{n-1}\}$被排序，且 s_i=max $\{ s_0, ..., s_i\}$）

例 13.2　冒泡排序

```
public static void sort(int[] a)
{ // POSTCONDITION: a[0] <= a[1] <= ... <= a[a.length-1];
  for (int i=a.length-1; i>0; i--)                    // step 1
    for (int j=1; j<=i; j++)                          // step 2
      if (a[j-1]>a[j]) swap(a,j-1,j);                 // step 3
    // INVARIANTS: a[i] <= a[i+1] <= ... <= a[a.length-1];
    //             a[i] >= a[j] for all j < i;
}
```

例 13.3　跟踪冒泡排序

图 13.2 记录了对数组{77, 44, 99, 66, 33, 55, 88, 22}进行冒泡排序的各个步骤。

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]
77	44	99	66	33	55	88	22
44	77						
		66	99				
			33	99			
				55	99		
					88	99	
						22	**99**
	66	77					99
		33	77				99
			55	77			99
					22	**88**	99
		33	66			88	99
			55	66		88	99
				22	**77**	88	99
					77	88	99
			22	**66**	77	88	99
		22	**55**	66	77	88	89
	22	**44**	55	66	77	88	99
22	**33**	44	55	66	77	88	99

图 13.2　跟踪冒泡排序

定理 13.1　冒泡排序算法是正确的

定理的证明过程参见问题 13.14 的参考答案。

定理 13.2　冒泡排序的时间复杂性是 $O(n^2)$

定理的证明过程参见问题 13.15 的参考答案。

13.3 选择排序

选择排序与冒泡排序存在一些相似之处。如图 13.3 所示。它也要对待排序的具有 n 个元素的序列进行 $n-1$ 趟扫描。在每趟扫描过程中，算法都从未排序的部分中挑选出最大的元素，并将它放置到正确的位置。这一算法比冒泡算法的效率更高一些，因为从未排序部分中挑选最大元素的过程中，算法不移动任何元素。在每一趟扫描中，它只是在找到了最大元素后才进行一次元素的位置交换操作。这一算法之所以被称为“选择排序”，正是因为它在每趟扫描过程中要从未排序的部分中选择出最大者，并把这一元素放置在正确的位置上。

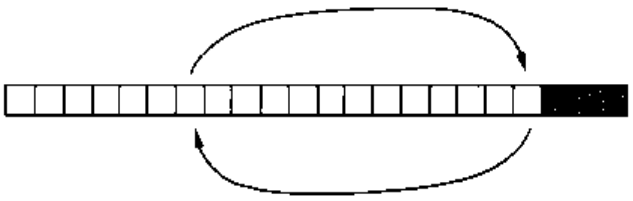


图 13.3 选择排序

算法 13.2 选择排序

（前置条件：$s=\{s_0, s_1, s_2, ..., s_{n-1}\}$是 n 个可排序元素的序列）

（后置条件：序列 s 被排序）

1. 令 i 从 $n-1$ 递减到 1，重复步骤 2~4。
2. 确定序列 $s=\{s_0, s_1, ..., s_i\}$中最大元素 s_j并记录 j。
3. 交换元素 s_i与 s_j。
4. （不变量：子序列$\{s_i, ..., s_{n-1}\}$被排序，且 $s_i=\max\{s_0, ..., s_i\}$）。

例 13.4 选择排序

```
public static void sort(int[] a)
{ for (int i=a.length-1; i>0; i--)
  { int j=0;
    for (int k=1; k<=i; k++)
      if (a[k] > a[j]) j = k;
    swap(a,i,j);
  }
}
```

例 13.5 跟踪选择排序

图 13.4 记录了对数组{77, 44, 99, 66, 33, 55, 88, 22}进行选择排序的各个步骤。

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]
77	44	99	66	33	55	88	22
		22					**99**
						88	99
55					**77**	88	99
			33	**66**	77	88	99
			55	66	77	88	99
		44	55	66	**77**	88	99
22	**33**	44	55	66	77	88	99

图 13.4 跟踪选择排序

标准分享网 www.bzfxw.com 免费下载

定理 13.3 选择排序算法是正确的

定理的证明过程参见问题 13.19 的参考答案。

定理 13.4 选择排序的时间复杂性是 $O(n^2)$

定理的证明过程参见问题 13.20 的参考答案。

尽管冒泡排序和选择排序的时间复杂性相同，但是在实际运行中，选择排序一般都要快一些。其中的原因从前面给出的两个算法执行的跟踪记录中也可以得到反映：冒泡排序共进行了 18 次元素交换操作，而选择排序只进行了 7 次。在两个需要交换位置的元素相隔很远的情况下，选择排序只需进行一次交换操作，而冒泡排序则需要进行很多次交换操作，选择排序正是凭借着这种能力才使得其运行效率优于冒泡排序。

13.4 插入排序

同前面介绍的选择排序与冒泡排序类似，插入排序也要对待排序的具有 n 个元素的序列进行 $n-1$ 趟扫描。如图 13.5 所示。在每趟扫描过程中，算法都将下一个待排序元素插入到该元素左边的子数组的合适位置上，使得这一子数组是排序的。在这种方法下，当数组的最后一个元素都被正确插入到其左边的子数组中后，整个数组就被排序了。

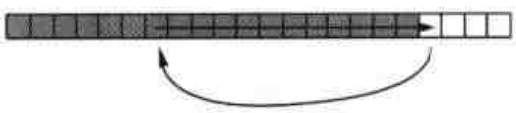


图 13.5 插入排序

算法 13.3 插入排序

（前置条件：$s=\{s_0, s_1, s_2, ..., s_{n-1}\}$是 n 个可排序元素的序列）

（后置条件：序列 s 被排序）

1. 令 i 从 1 递增到 $n-1$，重复步骤 2~4。
2. 将元素 s_i 保存在临时变量中。
3. 确定出使得条件 $s_j >= s_i$ 成立的最小的 j。
4. 将子序列$\{s_j, ..., s_{i-1}\}$后移一个位置到子序列$\{s_{j+1}, ..., s_i\}$的位置上。
5. 将保存在临时变量中的原来的 s_i 复制到 s_j 中。
6. （不变量：子序列$\{s_0, ..., s_i\}$被排序）。

例 13.6 插入排序

```
public static void sort(int[] a)
{ // POSTCONDITION: a[0] <= a[1] <= ... <= a[a.length-1];
  for (int i=1; i<a.length; i++)                    // Step 1
  { int temp=a[i], j;                               // Step 2
    for (j=i; j>0 && a[j-1]>temp; j--)              // Step 3
      a[j] = a[j-1];                                // Step 4
    a[j] = temp;                                    // Step 5
```

```
        // INVARIANT: a[0] <= a[1] <= ... <= a[i];
      }
    }
```

例 13.7 跟踪插入排序

图 13.6 记录了对数组{77, 44, 99, 66, 33, 55, 88, 22}进行插入排序的各个步骤。

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]
77	44	99	66	33	55	88	22
44	77						
	66	77	99				
33	44	66	77	99			
		55	66	77	99		
			77		**88**	99	
22	33	44	55	66	77	88	99

图 13.6 跟踪插入排序

定理 13.5 插入排序算法是正确的

定理的证明过程参见问题 13.23 的参考答案。

定理 13.6 插入排序的时间复杂性是 $O(n^2)$

定理的证明过程参见问题 13.24 的参考答案。

定理 13.7 对于一个已排序的序列，插入排序的时间复杂性是 $O(n)$

定理的证明过程参见问题 13.25 的参考答案。

13.5 SHELL 排序



图 13.7 SHELL 排序

定理 13.7 说明了这样一个事实：如果待排序的序列本身就是近似排序的，那么插入排序的时间复杂性将接近 $O(n)$。SHELL 排序就是利用了这样一个事实，它使得算法在一般情况下的时间复杂性优于 $O(n^{1.5})$。它反复地将插入排序作用在诸如$\{s_0, s_3, s_6, s_9, \ldots, s_{n-2}\}$和$\{s_1, s_4, s_7, s_{10}, \ldots, s_{n-1}\}$的等步长子序列（skip-subsequence）上。类似步长为 3 的子序列共 3 个，如图 13.7 所示。

算法 13.4 SHELL 排序

（前置条件：$s=\{s_0, s_1, \ldots, s_{n-1}\}$是 n 个可排序元素的序列）

（后置条件：序列 s 被排序）

1. 令 d=1；
2. 重复步骤 3 直到 $9d > n$ 为止；
3. 令 d=3d+1；
4. 重复步骤 5~6 直到 d=0；
5. 对于 s 的 d 个步长为 d 的子序列运用插入排序；

标准分享网 www.bzfxw.com 免费下载

6. 令 $d = d/3$;

例 13.8 SHELL 排序

假设序列 s 有 n=200 个元素，则算法第 2 步中需要循环 3 次，在这 3 次中，d 从 1 增加到 d=4、13 和 40。

步骤 4 中的第一次循环将对于 s 的 40 个步长为 40 的子序列分别运用插入排序，即：$\{s_0, s_{40}, s_{80}, s_{120}, s_{160}\}$，$\{s_1, s_{41}, s_{81}, s_{121}, s_{161}\}$，$\{s_2, s_{42}, s_{82}, s_{122}, s_{162}\}$，……，$\{s_{39}, s_{79}, s_{119}, s_{159}, s_{199}\}$。接下来，步骤 6 中将把 d 设为 13，则步骤 4 中的第二次循环将对于 s 的 13 个步长为 13 的子序列分别运用插入排序，即：$\{s_0, s_{13}, s_{26}, s_{39}, s_{52}, s_{65}, \ldots, s_{194}\}$，$\{s_1, s_{14}, s_{27}, s_{40}, s_{53}, s_{66}, \ldots, s_{195}\}$，……，$\{s_{12}, s_{25}, s_{38}, s_{51}, s_{64}, s_{77}, \ldots, s_{193}\}$。接下来，步骤 6 中将把 d 设为 4，则步骤 4 中的第三次循环将对于 s 的 4 个步长为 4 的子序列分别运用插入排序，即：$\{s_0, s_4, s_8, s_{12}, \ldots, s_{196}\}$，$\{s_1, s_5, s_9, s_{13}, \ldots, s_{197}\}$，$\{s_2, s_6, s_{10}, s_{14}, \ldots, s_{198}\}$和$\{s_3, s_7, s_{11}, s_{15}, \ldots, s_{199}\}$。再接下来，步骤 6 中将把 d 设为 1，则步骤 4 中的第四次循环将对于整个序列运用插入排序。整个过程将调用插入排序 58 次：其中，大小 n_1=5 的子序列共 40 次，大小 n_2=15 的子序列共 13 次，大小 n_3=50 的子序列共 4 次，大小 n_4=n=200 的子序列共 1 次。

初看起来，在 SHELL 排序中反复运用插入排序似乎要比直接对整个序列运用插入排序花费更多的时间。实际上，如果运用复杂性函数 n^2 直接计算一下例 13.8 中的比较次数就会得到：

$$40(n_1^2) + 13(n_2^2) + 4(n_3^2) + 1(n_4^2) = 40(5^2) + 13(15^2) + 4(50^2) + 1(200^2) = 53\,925$$

这个结果的确要比 $n^2 = 200^2 = 40\,000$ 更大。但是，经过步骤 4 的第一次循环，后续的子序列就成为近似排序的了。因此，实际的比较次数应该接近 n。所以，事实上的比较次数应该是：

$$40(n_1^2) + 13(n_2) + 4(n_3) + 1(n_4) = 40(5^2) + 13(15) + 4(50) + 1(200) = 1\,595$$

这个结果就比 40 000 要好许多了。

定理 13.8 SHELL 排序的时间复杂性为 $O(n^{1.5})$

请注意，对于 n=200，$n^{1.5} = 200^{1.5} = 2\,829$，这比起 $n^2 = 200^2 = 40\,000$ 要好得多。

例 13.9 SHELL 排序

```
public static void sort(int[] a)
{ int d=1, j, n=a.length;                    // step 1
  while (9*d<n)                              // step 2
    d = 3*d + 1;                             // step 3
  while (d>0)                                // step 4
  { for (int i=d; i<n; i++)                  // step 5
    { int temp=a[i];
      j=i;
      while (j>=d && a[j-d]>temp)
      { a[j] = a[j-d];
        j -= d;
```

```
        }
        a[j] = temp;
      }
      d /= 3;                                          // step 6
    }
  }
```

13.6 归并排序

归并排序是利用分治法来对序列进行排序的。首先，它不断地分割序列直到各子序列都成为单元素序列，然后，它又不断成对儿的合并各子序列直至恢复整个序列。在合并的过程中通过维护序列的顺序来保证合并后的序列是排序的。这样，当最后一次合并操作完成时，整个序列也就被排序了，如图13.8所示。

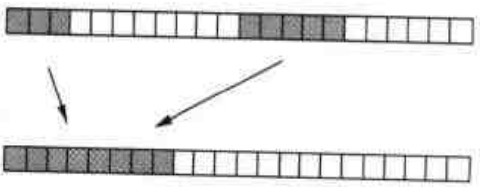


图13.8 归并排序

归并排序虽然完全可以实现为不断迭代的过程，但是一般还是将它实现为递归方法：首先将当前序列一分为二，然后再合并它们，在合并的同时进行排序。分割过程一直要继续到每个子序列都包含单个元素时才开始合并过程。

算法13.5 归并排序

（前置条件：$s=\{s_0, s_1, s_2, ..., s_{n-1}\}$是$n$个可排序元素的序列）

（后置条件：序列s被排序）

1. 如果$n>1$，重复步骤2~5；
2. 将序列s分割成两个子序列：$a=\{s_0, ..., s_{m-1}\}$和$b=\{s_m, ..., s_{n-1}\}$，其中，$m=n/2$；
3. 对a进行归并排序；
4. 对b进行归并排序；
5. 按照顺序合并子序列a和b得到s，符合s。

例13.10 归并排序

```
public static void sort(int[] a)
{ // POSTCONDITION: a[0] <= a[1] <= ... <= a[a.length-1];
  if (a.length>1) sort(a,0,a.length);                  // step 1
}

private static void sort(int[] a, int k, int n)
{ // PRECONDITIONS: 2 <= n <= k+n <= a.length;
  // POSTCONDITION: a[k] <= a[k+1] <= ... <= a[k+n-1];
  if (n<2) return;
  sort(a,k,n/2);
                                                       // steps 2 & 3
```

标准分享网 www.bzfxw.com 免费下载

```
  sort(a,k+n/2,n-n/2);                                    // step 4
  merge(a,k,n);                                           // step 5
}

private static void merge(int[] a, int k, int n)
{ // PRECONDITIONS: 1 <= n <= k+n <= a.length;
  //              a[k] <= a[k+1] <= ... <= a[k+n/2-1];
  //              a[k+n/2] <= ... <= a[k+n-1];
  // POSTCONDITION: a[k] <= a[k+1] <= ... <= a[k+n-1];
  int[] temp = new int[n];
  int i=0, lo=k, hi=k+n/2;
  while (lo<k+n/2 && hi<k+n)
    temp[i++] = ( a[lo]<=a[hi] ? a[lo++] : a[hi++] );
    // INVARIANT: temp[0..i] is sorted
  while (lo<k+n/2)
    temp[i++] = a[lo++];
    // INVARIANT: temp[0..i] is sorted
  while (hi<k+n)
    temp[i++] = a[hi++];
    // INVARIANT: temp[0..i] is sorted
  for (i=0; i<n; i++)
    a[k+i] = temp[i];
}
```

上面的程序段中的主 sort()方法通过调用另一个带起始元素下标 k 及子序列长度 n 的覆盖的 sort()方法实现对数组的排序。其中，带有 3 个参数的 sort()方法又分别对子序列的左半部分及右半部分进行归并排序，然后在合并起来，从而达到对整个子序列的排序。

merge()方法对子序列 s[k..m-1]及 s[m..k+n-1]的合并是通过一个临时数组完成的，其中，m 是子序列的中间元素的下标值 m=k+n/2。第一个 while 语句的每次循环中都拷贝一个数组元素：它拷贝的是 a[lo]和 a[hi]中的较小者。后置的递增运算符（++）修改被拷贝的元素下标。当前一半元素全部都拷贝完成后，另外两个 while 语句之一将把余下的元素拷贝到 temp[]中。最终，所有的元素又被拷贝回 a[]。

例 13.11 跟踪归并排序

图 13.9 记录了对数组{77, 44, 99, 66, 33, 55, 88, 22}进行归并排序的各个步骤。

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]
77	44	99	66	33	55	88	22
44	77						
		66	99				
	66	77					
						22	44
22	33	44	44	55	66	77	88

图 13.9 跟踪归并排序

图 13.10 说明了子数组是如何被拆分及合并的。

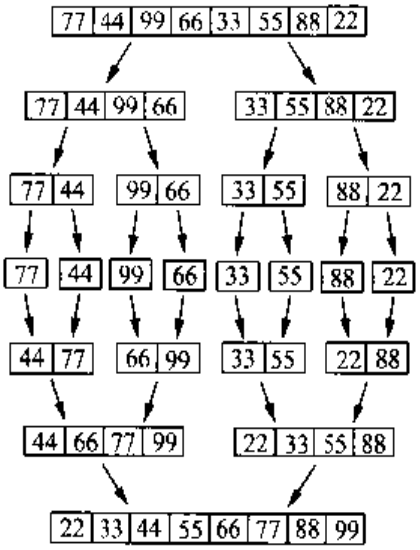


图 13.10 子数组的拆分及合并

定理 13.9 归并排序的时间复杂性是 $O(n \lg n)$

证明：通常，归并排序不断地将数组一分为二直至每一部分都成为单个元素，然后又不断地成对儿合并它们，直至最终合并成一个数组。这一过程从上面的这张图就可以看出。前一个过程的迭代数等于 n 能够被二分的次数，即 $\lg n-1$。第二个过程是第一个过程的反过程，因此它同样也包含 $\lg n-1$ 步。为此，整个算法包括 $O(\lg n)$步，而每一步需要进行 n 次元素的比较。所以，算法的元素比较总数是 $O(n \lg n)$。

定理 13.10 归并排序算法是正确的

证明：算法正确性证明过程的思路源于代码中的前置条件和后置条件。在主 `sort()`方法中，如果数组的长度为 0 或 1，则不必对它进行排序操作。否则，带有 3 个参数的 `sort()`方法的后置条件将确保数组在该方法被调用后被排序，因为传递给它的是整个数组。这个后置条件与 `merge()`方法的后置条件是相同的，而 `merge()`方法是在最后被调用的，因此，剩下的工作就是证明 `merge()`方法的后置条件是正确的。

`merge()`方法的后置条件的正确性取决于其中的 3 个循环不变量，因为一旦这 3 个循环完成，临时数组 `temp[]`将被排序而后按照同样的顺序被重新拷贝回数组 `a[]`。因此，剩下的工作就是证明这 3 个循环不变量的正确性。

假设在这 3 个循环中，存在某个元素 y，它被拷贝到数组 `temp[]`中且比某个先于它拷贝的元素 x 小。那么，元素 x 必定是从另外一个子数组中被拷贝到 `temp[]`中去的，因为这两个子数组都是被各自事先排序的，如果 x 与 y 同处于一个子数组中，因为 x 比 y 大，它就不可能后于 y 被拷贝（如图 13.11 所示）。也就是说，x 的拷贝是在第一个循环中进

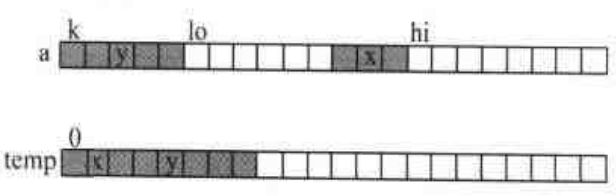


图 13.11 数组排序示例

标准分享网 www.bzfxw.com 免费下载

行的。不失一般性，我们假设 y 处于数组的前一半，x 处于数组的后一半。现在有 $y<x$，而这两个子数组都是排序的，因此从 a[k]到 y 的所有元素都应该比 x 小。但是下面的赋值语句：

```
temp[i++] = ( a[lo]<=a[hi] ? a[lo++] : a[hi++] );
```

总是先拷贝小的元素，因此它在拷贝 x 之前必然已经将 a[k]到 y 的所有元素都拷贝到 temp[]中了，但是这与假设矛盾，因此，假设不成立，循环不变量成立。

通过运用分治法的策略，归并排序获得的算法复杂性是 $O(n\lg n)$的，这比前面介绍的算法的 $O(n^2)$复杂性在效率上有了很大的提高。分治法策略的主要步骤是：

1．将序列分割成两个子序列；

2．分别独立地对两个子序列进行排序；

3．将两个子序列重新合并起来。

归并算法在完成上面第 1 步时采用了最简单平衡分割方法：从序列的中点分割序列。如果分割的位置不同，将得到不同的排序算法。分治法还被用于折半查找算法（参见算法 2.2）。

最简单的非平衡分割方法是从序列的倒数第二个元素进行分割，让第二个子序列只包含一个元素。这种分割方式下得到的算法就是递归版本的插入排序算法（参见习题 13.22）。

另外一种非平衡分割方法是将序列中的最大元素作为第二个子序列分割出来，其他元素都放到第一个子序列中。这种分割方式下得到的算法就是递归版本的选择算法。（参见习题 13.18）注意到，这种方式下，第三个步骤将非常简单，只需要将最大元素追加到第一个子序列最后即可。

第四种分割方法是使得分割后的第一个子序列中的所有元素都比第二个子序列中的小。在前面所述的递归版本的选择算法中，这一性质是成立的。然而，还可以做到既满足这个性质，又使得两个子序列具有相同的大小。这就是另外一个 O(n lg n)复杂性的排序算法：快速排序算法。

13.7　快速排序

快速排序类似于归并排序：它也是递归的，它也需要额外的包含多个循环的辅助函数，它也是 $O(n \lg n)$的。但是，在大多数情况下，它比归并排序要快，如图 13.12 所示。

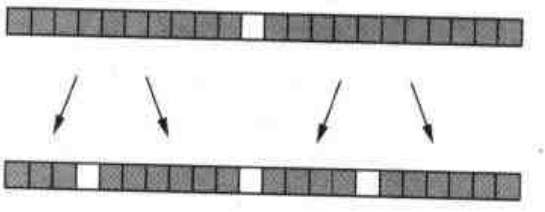


图 13.12　快速排序

快速排序在分割数组时选择了一个具有下列性质的元素 x（称为基准元素）作为分割点：所有处在 x 左边的元素都不比 x 大，所有处在 x 右边的元素都不比 x 小。这一性质就确保了元素 x 处在它应该在的“正确的”位置上。算法就是依据这样的思想继续分别对子序列进行分割的。这一过程非常适合于用递归方法实现，而且速度非常快。

算法 13.6　快速排序

（前置条件：$s=\{s_0, s_1, s_2, ..., s_{n-1}\}$是 n 个可排序元素的序列）

（后置条件：序列 s 被排序）

1. 如果 $n>1$，重复步骤 2~5；
2. 将快速分割算法（参见算法 13.7）运用于 s，获得当前的基准元素 s_i；
3. （循环不变量：基准元素 s_i 处于排序后的正确位置）
4. 对 $a=\{s_0, s_1, ..., s_{i-1}\}$进行快速排序；
5. 对 $b=\{s_{i+1}, s_{i+2}, ..., s_{n-1}\}$进行快速排序；

算法 13.7 快速分割算法

（前置条件：$s=\{s_k, s_{k+1}, s_{k+2}, ..., s_{k+n-1}\}$是 n 个可排序元素的序列）

（后置条件：$k\leqslant j<n$ 且对于 $k\leqslant p<j<q<k+n$ 有 $s_p\leqslant s_j\leqslant s_q$）

1. 令 $x=s_k$（基准元素）；
2. 令 $i=k$，$j=k+n$；
3. 当 $i<j$ 时，重复步骤 4~7；
4. 重复递增 i，只要条件 $s_i<x$ 成立；
5. 重复递减 j，只要条件 $s_j>x$ 成立；
6. 交换 s_i 和 s_j；
7. （循环不变量：对于 $j<q<k+n$ 及 $k\leqslant p<i$，有 $s_p\leqslant x\leqslant s_q$）；
8. 交换 s_k 和 s_j；
9. （循环不变量：对于 $k\leqslant p<j<q<k+n$ 有 $s_p\leqslant s_j\leqslant s_q$）

例 13.12 快速排序

```
public static void sort(int[] a)
{ // POSTCONDITION: a[0] <= a[1] <= ... <= a[a.length-1];
  if (a.length>1) sort(a,0,a.length);                          // step Q.S.1
}

public static void sort(int[] a, int k, int n)
{ // PRECONDITIONS: 0 <= k <= k+n <= a.length;
  // POSTCONDITION: a[k] <= a[k+1] <= ... <= a[k+n-1];
  if (n<2) return;
  int pivot = a[k];                                            // step Q.P.1
  int i = k;
  int j = k+n;
  while (i < j)                                                // step Q.P.3
  { while (i+1<k+n && a[++i]<pivot) ;                          // step Q.P.4
    while (a[--j]>pivot) ;                                     // step Q.P.5
    if (i < j) swap(a,i,j);                                    // step Q.P.6
    // INVARIANT: a[p]<=pivot<=a[q] for k<=p=<i<j=<q<k+n       // step Q.P.7
  }
  swap(a,k,j);                                                 // step Q.P.8
```

标准分享网 www.bzfxw.com 免费下载

```
    // INVARIANT: a[p]<=a[j]<=a[q] for k<p<j<q<k+n          // step Q.P.9
    sort(a,k,j-k);                                          // step Q.S.4
    sort(a,j+1,k+n-j-1);                                    // step Q.S.5
  }
```

例 13.13 跟踪快速排序

图 13.13 记录了对数组{77, 44, 99, 66, 33, 55, 88, 22}进行快速排序的各个步骤。

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]
77	44	99	66	33	55	88	22
		22					99
55					**77**		
			33	66	77		
33			**55**		77	**88**	
	22	44	**55**		77	**88**	
22	**33**		55		77	88	**99**
22	**33**	**44**	55	**66**	77	88	99

图 13.13 跟踪快速排序

算法 13.7 中选取的基准元素是序列的最后一个元素。当然也可以选取序列的第一个元素或中间元素。如果选取上述 3 个元素中的中间元素，算法的性能会稍好些。

Java 中的 `Arrays.sort()` 方法实现的就是快速排序。当 n<40 时，它选取的基准元素是$\{s_0, s_{n/2}, s_{n-1}\}$三者的中间值；当 n>40 时，将选取 9 个间隔相当的元素的中间值。在 n<7 时，它还会切换为插入排序（参见算法 13.3）。

定理 13.11 快速排序在最好情况下的时间复杂性是 $O(n \lg n)$

证明：最好的情况是当序列的值满足随机的均匀分布，这样，每次调用快速分割算法将比较平衡地对序列进行分割。在这种情况下，每次递归地调用快速排序都会把序列分割成长度近似相等的两个子序列。同折半查找算法（参见算法 2.2）及归并排序（参见算法 13.5）一样，重复这一过程直至得到长度为 1 的子序列需要 $\lg n$ 步，如图 13.14 所示。因此，总共需要对快速分割算法调用 $O(\lg n)$次，而快速分割算法的运行时间为 $O(n)$。所以，快速排序算法的总的时间复杂性为 $O(n \lg n)$。



图 13.14 快速排序的复杂性

定理 13.12 快速排序在最坏情况下的时间复杂性是 $O(n^2)$

证明：最坏的情况是当序列已经被排序了（或是反序的）。在这种情况下，快速排序又总是选取最后一个元素（或当序列呈反序时选取第一个元素），对序列的分割总是最不平衡的情况：一个子序列包含 n–2 个元素，而另一个子序列只包含一个元素。在这样的情况下，要使得每一部分都只包含长度为 1 的子序列，需要进行 $O(n)$次。因此，总共需要对快速分

割算法调用 $O(n)$次，而快速分割算法的运行时间为 $O(n)$。所以，快速排序算法的总的时间复杂性为 $O(n^2)$。

注意到，在最坏的情况下，快速排序同选择排序（算法 13.2）相当，因为每次对快速分割算法的调用总是选取了当前子序列中的最大值。因此，实际上，定理 13.12 是定理 13.4 的推论。

定理 13.13　快速排序在平均情况下的时间复杂性是 $O(n \lg n)$

这一定理的证明过程超出了本书的范围。

定理 13.14　快速排序是正确的

证明：while 循环中的不变量表明，在元素 a[i]左边的元素都不比基准元素大，在元素 a[j]右边的元素都不比基准元素小。这一事实的正确性是因为，在 a[i]的左边如果有比基准元素大的元素，它就会与在 a[j]右边的某个比基准元素小的元素交换位置。对于反序情况，也有类似情况（在 a[j]的右边如果有比基准元素小的元素，它就会与在 a[i]左边的某个比基准元素大的元素交换位置。）当循环结束时有 j≤i，而且此时，所有比基准元素大的元素已经被移到 a[i]的右边，所有比基准元素小的元素都被移到了 a[i]的左边。再经过第 8 步的元素交换可知，所有比 a[i]大的元素都在 a[i]的右边，所有比 a[i]小的元素都在 a[i]的左边。于是得到快速分割算法的第 7 步的不变量，也就是快速排序算法的第 3 步的不变量。继续将这一过程分别在左右两个子序列上递归地作用下去，就可以完成对整个序列排序。

13.8　堆排序

根据定义，堆本身就是部分排序的，因为从堆的根节点到叶节点的每一条路径上的元素都是排序的。（参见第 12 章）利用这一思想得到一种有效的通用排序算法，即堆排序。同其他的排序算法一样，堆排序也是作用于数组或向量的。不同的是，堆排序的底层使用的是数组的堆表示（二叉树）形式。

类似于和归并排序和快速排序，堆排序也需要在 sort()方法中调用额外的辅助函数，它也是 $O(n \lg n)$的。但是，同归并排序及快速排序不同的是，它不是递归算法。

从算法的本质上看，堆排序实际上是首先将 n 个元素装入堆，然后再把它们取出。根据定理 12.1，每个元素的装入时间是 $O(\lg n)$的，取出时间是也是 $O(\lg n)$ 的，因此对于整个算法来说，它需要处理 n 个元素的装入和取出，复杂性就是 $O(n \lg n)$的。

算法 13.8　堆排序

（前置条件：$s=\{s_0, s_1, s_2, \ldots, s_{n-1}\}$是 n 个可排序元素的序列）

（后置条件：序列 s 被排序）

1. 令 i 从 $n/2-1$ 递减到 0，重复步骤 2~3；
2. 让 Heapify 算法作用在子序列$\{s_i, s_{i+1}, \ldots, s_{n-1}\}$上；
3. （循环不变量：s 中的所有由根节点到叶节点的路径均是非递增的）；
4. 令 i 从 $n-1$ 递减到 1，重复步骤 5~7；
5. 交换 s_i 和 s_0；

标准分享网 www.bzfxw.com 免费下载

6.（循环不变量：子序列$\{s_i, s_{i+1}, ..., s_{n-1}\}$已被排序）

7. 让 Heapify 算法作用在子序列$\{s_0, s_1, ..., s_{i-1}\}$上；

算法 13.9 Heapify 算法

（前置条件：$ss=\{s_i, s_{i+1}, ..., s_{j-1}\}$是长度为 $j-i$ 的子序列，其中，$0 \leqslant i < j \leqslant n$，且子序列$\{s_{i+1}, ..., s_{j-1}\}$及$\{s_{i+2}, ..., s_{j-1}\}$都具有堆性质）

（后置条件：序列 ss 本身也具有堆性质）

1. 令 $t = s_{2i+1}$；

2. 令 $s_k = \max\{s_{2i+1}, s_{2i+2}\}$，$k=2i+1$ 或 $2i+2$，即两个孩子节点中大者的下标值；

3. 如果 $t < s_k$，重复步骤 4~6；

4. 令 $s_i = s_k$；

5. 令 $i = k$；

6. 如果 $i < n/2$ 且 $s_i < \max\{s_{2i+1}, s_{2i+2}\}$，重复步骤 1~4；

7. 令 $s_k = t$；

这里的算法同第 12 章中的算法存在两个方面的差异：第一，这里的堆中的元素的顺序是反序排列的，即从根节点到叶节点是非递增序列。第二，这里采用的是从 0 开始的下标编码方式。反序排列确保了 Heapify 算法总是将子序列中的最大元素置于根节点的位置。采用从 0 开始的下标编码方式使得本算法与其他排序算法在这一点上是相符的，当然，代价就是使得算法的代码略为复杂了些。

例 13.14 堆排序

```
public static void sort(int[] a)
{ // POSTCONDITION: a[0] <= a[1] <= ... <= a[a.length-1];
  for (int i=a.length/2-1; i>=0; i--)              // step 1
    heapify(a,i,a.length);                         // step 2
  for (int i=a.length-1; i>0; i--)                 // step 4
  { swap(a,0,i);                                   // step 5
    heapify(a,0,i);                                // step 7
  }
}

private static void heapify(int[] a, int i, int j)
{ int tmp=a[i], k=0;                               // step 1
  while (2*i+1<j)
  { k=2*i+1;
    if (k+1<j && a[k+1] > a[k]) ++k;  // a[k] is the larger child
    if (tmp>=a[k]) break;                          // step 3
    a[i] = a[k];                                   // step 4
    i = k;                                         // step 5
  }
```

```
    a[k] = tmp;                                    // step 7
  }
```

注意到，这里的heapify()方法与例12.9中的heapifyDown()方法是等价的。

例13.15 跟踪堆排序

图13.15记录了对数组{77, 44, 99, 66, 33, 55, 88, 22, 44}进行堆排序的各个步骤。

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	A[6]	a[7]	a[8]
77	44	99	66	33	55	88	22	44
	66		**44**					
99	66	**88**	44	33	55	**77**	22	44
44								**99**
88	66	**77**	44	33	55	**44**	22	
22							**88**	
77	66	**55**	44	33	**22**	44		
44						**77**		
66	**44**	55	44	33	22			
22					**66**			
55	44	**22**	44	33				
33				**55**		77		
44	44	22	**33**					
33			**44**					
44	**33**	22						
22		**44**						
33	**22**							
22	33							

图13.15 跟踪堆排序

图13.15中的每个阴影部分表示的都是对heapify()方法的调用结果。这一过程在步骤2中进行一次，在步骤7中还要进行n-2次。

sort()方法首先对数组作适当转化，以使得其对应的完全二叉树被转化成堆。这是通过将heapify()方法作用在每棵非简单子树上完成的。这里，非简单子树是指子树的元素数目大于1，且子树的根节点所处的层高于叶节点所处的层。在数组中，叶节点被存储在位置a[n/2]到a[n]中。因此，sort()方法中的第一个for循环将heapify()方法作用于元素a[n/2-1]到a[0]（a[0]是对应的二叉树的根节点）。其结果是一个具有堆性质的数组，如图13.16所示。

至此，sort()方法中的主for循环（即第二个for循环）将进行n-1次循环。每次循环完成下列两件事：第一，交换根元素和元素a[i]；第二，将heapify()方法作用到元素a[0:i-1]对应的子树上。这棵子树中的元素有一些还是为排序的。每次循环中，在swap()方法执行之前，子数组a[0:i]具备堆性质，因此元素a[i]是该子数组中的最大元素。这就是说，swap()方法将元素a[i]放到了它应该在的“正确”位置上。

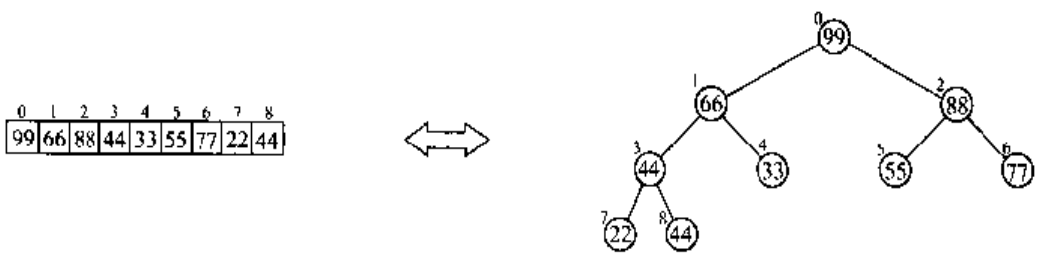


图13.16 堆排序

标准分享网 www.bzfxw.com 免费下载

sort()方法中的主 for 循环的前 7 次循环的效果如下面的 7 张图所示。数组（及其对应的二叉树）被分成两部分：第一部分是具有堆性质的子数组 a[0:i-1]，第二部分是余下的部分 a[i:n-1]，其中的元素都是处于它们应该处在的“正确”位置。在图 13.17 中，第二部分用阴影表示。主 for 循环的每次循环都不断地减小第一部分的大小，增大第二部分的大小。因此，当所有循环结束后，第一部分将为空，而第二部分就是整个数组了。从上面的分析可以看出，堆排序算法是正确的。

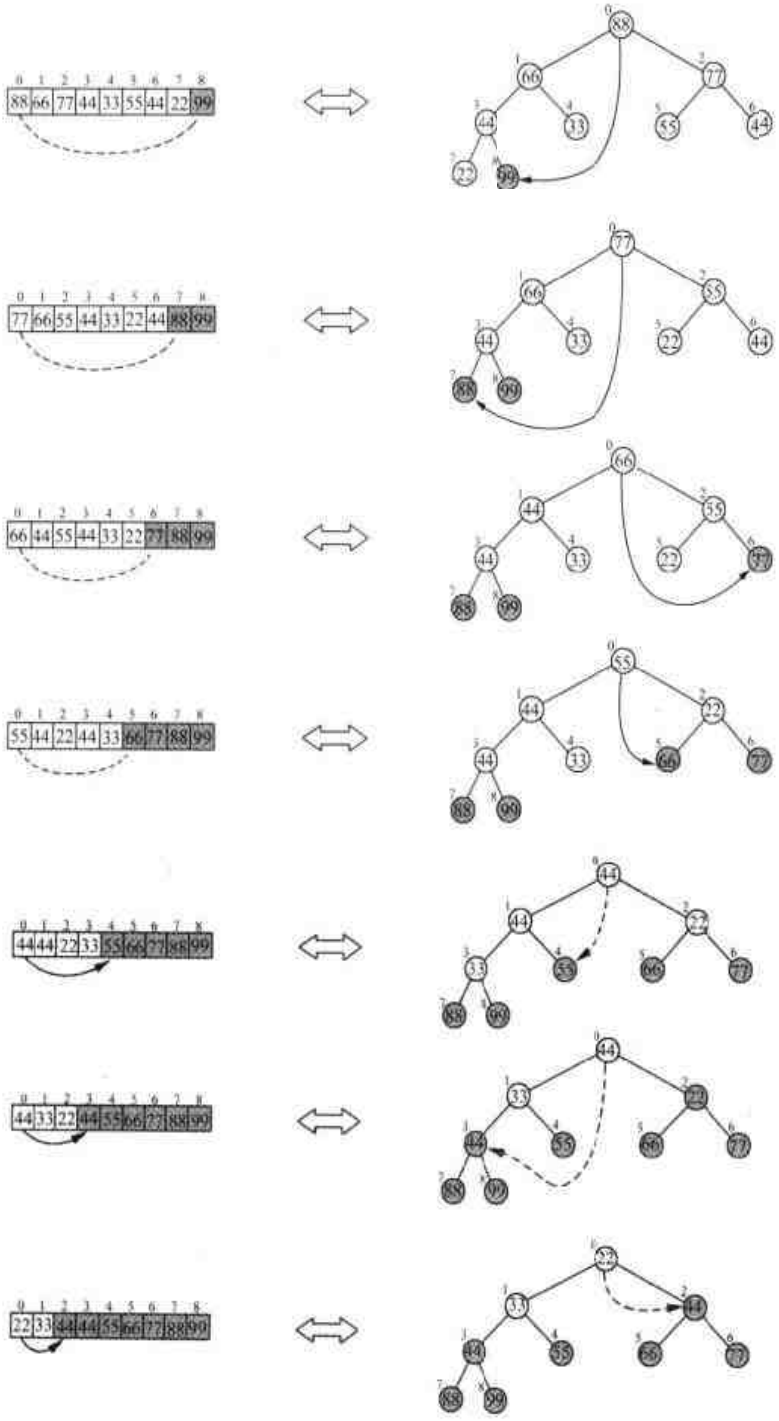


图 13.17　堆排序示例

定理 13.15　堆排序是正确的

参见习题 13.31 的参考答案。

定理 13.16 堆排序的时间复杂性是 $O(n \lg n)$

证明：对 heapify()方法的每一次调用将最多进行 lg n 步，因为该方法的迭代过程是基于从当前节点到叶节点的路径，而对于一个基于 n 个节点的完全二叉树来说，这样的路径最长不过 lg n。heapify()方法在第一个 for 循环中被调用了 $n/2$ 次，在第二个 for 循环中被调用了 $n-1$ 次，一共不超过 $(3n/2)\lg n$ 次，复杂度与 $n \lg n$ 相当。

假如我们将排序算法假想成是一种“元素流动的过程”，在这个过程中，各元素按照随机的顺序“流入”到一个数组中，然后按照排序好的顺序从数组中“流出”。在这个背景下，可以把堆排序看成是选择排序和插入排序的某种有效折衷，而选择排序和插入排序则分别是两个极端情况。选择排序将所有的排序操作都放到“流出过程”中完成，而元素在数组中的存储则是无序的，它完全由随机的“流入过程”决定。插入排序则将所有的排序操作都放到“流入过程”中完成，元素在进入数组存储后就是有序的，其流出顺序与存储顺序是对应的。与前面两种算法不同，堆排序在“流入过程”中只进行“部分排序”，而把剩下来的排序工作放到“流出过程”中完成。这种折衷带来的直接利益就是：堆排序的算法时间复杂性是 $O(n \lg n)$而不是 $O(n^2)$。

13.9 比较排序的速度限制

定理 13.17 通过比较数组元素大小对数组进行重新组织而达到排序目的的排序算法，其最坏情况的时间复杂性函数不可能优于 $O(n \lg n)$。

证明：可以用决策树来表示对大小为 n 的数组进行排序的算法的所有可能输出。由于算法是通过比较数组元素后重组数组达到排序目的的，因此，这棵决策树的每个节点的形式都是条件表达式(a[i]<a[j])。每个这样的条件表达式的可能结果有两个(true 或 false)，因此，这棵决策树是一棵二叉树。又由于这棵树必须覆盖所有可能的输出结果，它至少要有 $n!$个叶节点。根据推论 10.3，决策树的高度至少为 $\lg(n!)$。在最坏情况下，比较算法所作比较的次数应与决策树的树高相等。因此，算法在最坏情况下的复杂性函数为 $O(\lg(n!))$。

根据 Stirling 公式（参见附录中的定理 B.11），有

$$n! = \sqrt{2n\pi}\left(\frac{n}{e}\right)^n$$

则

$$\log_2(n!) = \log\left(\sqrt{2n\pi}\left(\frac{n}{e}\right)^n\right) \approx \log_2(n^n) = n\log_2 n$$

所以，算法最坏情况的时间复杂性函数为 $O(n \lg n)$。

定理 13.17 只对比较排序有效。所谓比较排序就是通过比较元素的值，并根据比较

标准分享网 www.bzfxw.com 免费下载

结果改变元素的相对位置来达到排序目的算法。本章前面介绍的所有算法都是比较排序算法。除了比较排序算法以外，还有一些其他类型的排序算法，我们将在本章的后续内容中讨论。

13.10 基数排序

基数排序的前提是待排序序列的元素类型是固定长度的词典数组，也就是说，是字符串类型或整数类型。令 r 是数组元素的“基数”（例如，对于 ASCII 字符串来说 r=26，对于十进制整数来说 r=10，对于二进制串来说，r=2）。令 w 是词典数组的固定长度。例如，对于美国的社会保险号来说，d=10 而 w=9。

例 13.16 根据 ISBN 号对书籍排序

从 70 年代起，每一本正式出版的书籍都有一个惟一的所谓“国际标准书号”（International Standard Book Number－ISBN），它通常在书的封底的下方。比如，一本书的 ISBN 号为 0071361286（它通常表示成 0-07-136128-6 以区分组成书号的 4 个域）。书号的最后位是校验位，最后一位的数字是根据前面 9 个数字计算出来的。由于它可以是 10 个数字之一或者是字母 X，可知基数 r=11，数字个数 d=10。

算法 13.10 基数排序

（前置条件：$s=\{s_0, s_1, s_2, ..., s_{n-1}\}$是 n 个基于为 r，宽度为 w 的整数或字符串的序列）

（后置条件：序列 s 依词典序被排序）

1. 令 d 从 0 递增到 w–1，重复步骤 2；
2. 用一个稳定排序算法来对序列 s 进行排序，排序仅仅在第 d 位上进行；

我们说一个排序算法是“稳定的”是指，对于相等值的元素来说，算法不会改变这些元素之间的相对位置。例如，插入排序就是稳定排序算法。

例 13.17 根据基数排序对 ISBN 号进行排序

图 13.18 给出了用基数排序对 12 个 ISBN 号进行排序的前三次循环的情况：



图 13.18 基数排序示例

下面我们通过例子可以看到，算法只有具备了“稳定性”，才能使得算法执行过程中的前面循环的排序效果得到继承和保护。例如，在第一次循环完成后，8838650527 排在 0830636528 的前面，因为在最低位数字上有 7<8。在倒数第二个低位上（即当 d=1 时），两者的数字相同，都是 2。因此，在第二次循环中，由于只对 d=1 位排序，这两个元素被认为是相等的。但是由于 27<28，它们必须保存原来的相对次序。算法的“稳定性”就是要保证这一点。

已经处理过的列在图 13.18 中用阴影部分表示。经过三次循环后，最右边的三列已经被排序成：109<13X<286<373<453 等等。（X 代表 10，因此 13X 就是数字 130+10=140）

等到余下的 7 次循环完成后，序列应该被排序成如图 13.19 所示。

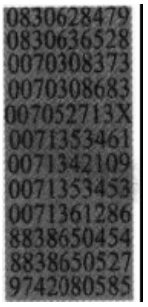


图 13.19 基数排序示例结果

例 13.18 基数排序

下面的方法假设常量 RADIX 和 WIDTH 已经被定义了。例如，对于 int 型数组：

```
public static final int RADIX=10;
public static final int WIDTH=10;

public static void sort(int[] a)
{ // POSTCONDITION: a[0] <= a[1] <= ... <= a[a.length-1];
  for (int d=0; d<WIDTH; d++)                          // step 1
    sort(a,d);                                         // step 2
}

private static void sort(int[] a, int d)
{ // POSTCONDITION: a[] is sorted stably on digit d;
  int[] c = new int[RADIX];
  for (int i=0; i<a.length; i++)
    ++c[digit(d,a[i])];  // tally the values in a[p..q]
  for (int j=1; j<RADIX; j++)
    c[j] += c[j-1];  // c[j] == num elts in a[] that are <= j
  int[] tmp = new int[a.length];
  for (int i=a.length-1; i>=0; i--)
    tmp[--c[digit(d,a[i])]] = a[i];
  for (int i=0; i<a.length; i++)
  a[i] = tmp[i];
}

private static int digit(int d, int x)
{ // e.g., digit(3,1234567890) returns 7;
  return x/(int)Math.pow(10,d)%RADIX;
```

标准分享网 www.bzfxw.com 免费下载

```
    }
}
```

上面的代码中的第二个 *sort* 方法又被称为计数排序（*counting sort*）或计算排序（*tally sort*）。

定理 13.18 基数排序的时间复杂性为 $O(n)$

证明：算法共需循环 WIDTH 次，在每次循环中对 n 个元素处理 3 次。因此算法运行时间正比于 WIDTH×n，其结果是一个常量。

尽管从理论上来讲，复杂性为 $O(n)$的算法应该比复杂性为 $O(n \lg n)$的快，但是在实际中，基数排序却很少比 $O(n \lg n)$算法（如归并排序、快速排序和堆排序）快。这是因为基数排序还需要较多额外的操作，如各位数字的提取，数组的拷贝等等。

13.11 吊桶排序

吊桶排序也是“分配”排序。它根据一些粗粒度的排序准则将元素分配到多个“吊桶”中，然后再利用另外的排序算法来对每个吊桶中的元素进行排序。与快速排序相似，吊桶 i 中的所有元素都大于或等于吊桶 i–1 中的所有元素，同时又小于或等于吊桶 i+1 中的所有元素。可以这样看，快速排序是将元素分配到两个吊桶中进行排序的，而吊桶排序是将元素分配到 n 个吊桶中进行排序的，如图 13.20 所示。

算法 13.11 吊桶排序

（前置条件：$s=\{s_0, s_1, s_2, ..., s_{n-1}\}$是 n 个可排序元素的序列，且已知其中最小元素为 min，最大元素为 max）

（后置条件：序列 s 被排序）

1. 初始化包含 n 个吊桶（集合）的数组；
2. 对于序列中的每个元素 s_i，重复步骤 3；
3. 将元素 s_i 插入吊桶 j，其中 $j=\lfloor r\ n \rfloor$，$r=(s_i-\min)/(\max+1-\min)$；
4. 分别对各个吊桶进行排序；
5. 令 $j=0$ 到 n–1，重复步骤 6；
6. 将吊桶 j 中的元素顺序放回到序列 s 中。

例 13.19 用吊桶排序对社会保险号进行排序

假设现在有 1 000 个 9 位的社会保险号。建立 1000 个整型数组，并通过公式 $j=\lfloor r\ n \rfloor$，$r=(s_i-\min)/(\max+1-\min)=(s_i-0)/(10^9+1-0)\approx s_i/10^9$ 将这 1 000 个社会保险号分配到这些数组中去。例如，社会保险号 666666666 将被插入到第 j 号吊桶，其中 $j=\lfloor r\ n \rfloor=\lfloor(666666666/10^9)(10^3)\rfloor=\lfloor 666.666666 \rfloor=666$。同样，社会保险号 123456789 将被插入到第 123 号吊桶，社会保险号 66654321 将被插入到第 666 号吊桶中。

然后，分别对每个吊桶中的元素进行排序。由于每个吊桶中的元素的平均长度为 1，因此排序算法的选取将不会影响到算法的运行时间。

最后，从第 0 号吊桶开始，将元素一一拷贝回序列 s 中。

例 13.20　吊桶排序

```
public static void sort(int[] a)
{ // POSTCONDITION: a[0] <= a[1] <= ... <= a[a.length-1];
  int min=minimum(a);
  int max=maximum(a);
  int n=a.length;
```

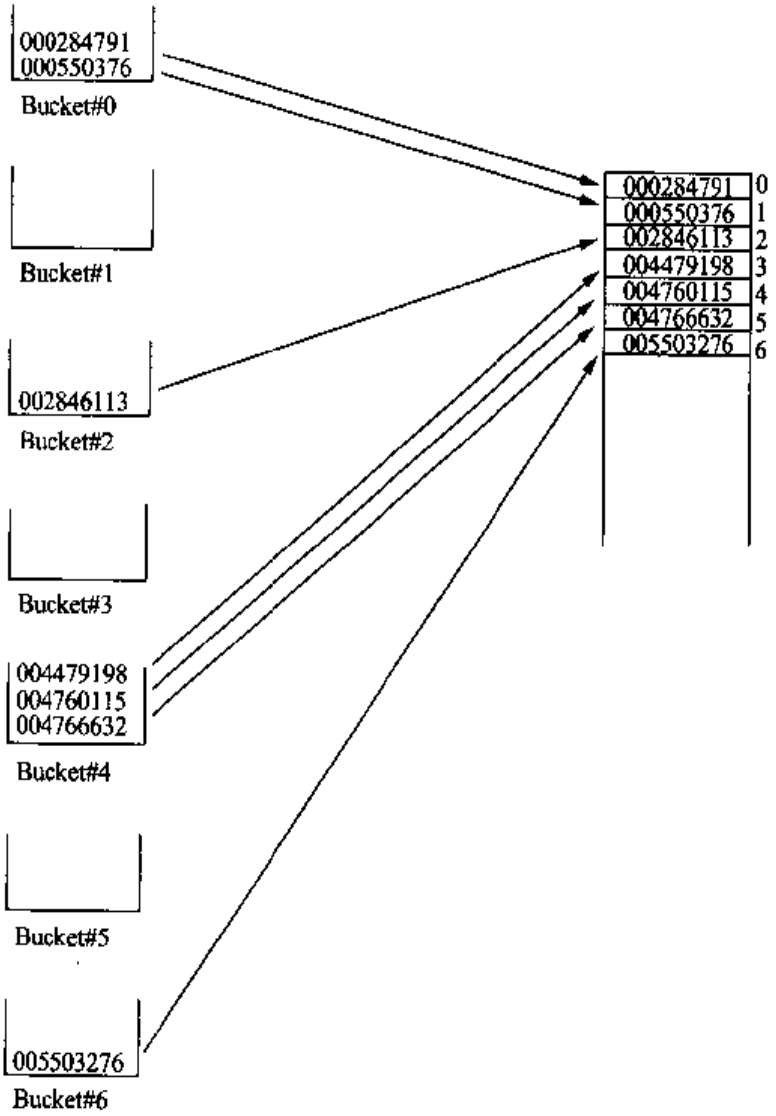


图 13.20　吊桶排序

```
Bucket[] bucket = new Bucket[n];                    // step 1
for (int j=0; j<n; j++)
  bucket[j] = new Bucket();
for (int i=0; i<n; i++)                             // step 2
```

标准分享网 www.bzfxw.com 免费下载

```
    { int j = n*(a[i]-min)/(max+1-min));
      bucket[j].add(a[j]);                                  // step 3
    }
    int i=0;
    for (int j=0; j<n; j++)
    { Bucket bucketj=bucket[j];
      bucketj.sort();                                       // step 4
      for (int k=0; k<bucketj.size(); k++)                  // step 5
        a[i++] = bucketj.get(k);
    }
  }
```

上面这段程序要求实现下面的接口：

```
public interface Bucket
{

  public void add(int x);  // 将 x 加入到吊桶中
  public int get(int k);    // 返回吊桶中第 k 个元素
  public int size();       // 返回吊桶中元素个数
}
```

此外，还需要实现下面两个局部方法：

```
public int minimum(int[ ] a);   // 返回 a[ ]中的最小元素
public int maximum(int[ ] a);   // 返回 a[ ]中的最大元素
```

定理 13.19 吊桶排序的时间复杂性是 $O(n)$

证明：算法包含 3 个并行的循环，每个循环迭代 n 次。最后一个循环中还包含一个平均迭代次数为 1 的内循环。minimum 方法和 maximum 方法也都各包含 n 步。因此，算法执行的总步骤数正比于 $5n$。

类似于基数排序，复杂度为 $O(n)$的吊桶排序在实际应用中比复杂度为 $O(n \lg n)$的算法要慢得多，这是由于吊桶算法本身复杂的额外开销所至。

复 习 题

13.1 冒泡排序执行很慢的原因是什么？

13.2 在定理 13.2 的证明中有“冒泡排序共进行 $n(n-1)/2$ 次比较”，为什么进而能够得出“算法的时间复杂度函数是 $O(n2)$的”？

13.3 为什么复杂度为 $O(n)$的排序算法（基数排序、吊桶排序）的执行速度比复杂度为 $O(n \lg n)$的排序算法（归并排序、快速排序和堆排序）的执行速度还要慢？

13.4 归并排序使用了一个通用的方法即分治法来对数组进行排序：它将数组分成很多小块，并在每一小块上循环调用自己。请问其他还有哪些算法用到了分治法？

13.5 哪个排序算法对于链表的排序和对于数组的排序同样好？

13.6 哪个排序算法的最坏情况复杂度不同于平均情况复杂度？

13.7 哪个排序算法的最好情况复杂度不同于平均情况复杂度？

13.8 为什么一个递归排序算法的非递归版本通常要高效一些？

13.9 快速排序与归并排序有哪些相像之处？

13.10 在什么样的条件下，归并排序比另外两个复杂度为 $O(n \lg n)$的算法好？

13.11 在什么样的条件下，快速排序会类似于选择排序？

13.12 在什么样的条件下，快速排序比另外两个复杂度为 $O(n \lg n)$的算法好？

13.13 堆排序与选择排序、插入排序有哪些相像之处？

13.14 Java 的 API 使用的是哪种排序算法来实现其 `java.utils.Arrays.sort()`方法？

13.15 如果一个排序算法能够保持相等元素在排序前后的相对次序，我们就说它是稳定的。请问哪些排序算法不是稳定的？

13.16 本章所涉及的 9 种排序算法中，哪些需要额外的数组空间？

13.17 本章所涉及的 9 种排序算法中，哪些特别适合于对以外部文件形式存在的记录进行排序？

13.18 我们说归并排序是可并行的，其含义是指在一台支持多处理器并行的计算机上，排序中各部分可以并行执行，而不管其他部分的执行情况。这对于归并排序来说是可行的，因为对数组各部分的分割及合并可以独立地进行，而不必关心其他部分。本章所涉及的其他算法还有哪些是可并行的？

13.19 假想有这样一个 Web 网站，网站上有一些 Java Applet，每个 Applet 都通过动画表示一个算法在数组 `a[]`上的执行过程，数组 `a[]`的大小为 256，元素随机取自 0.0 到 1.0 区间。对于算法主循环的每一次迭代，动画将显示一个 256 个点(x, y)的二维图，每个点对应了数组的一个元素，$x = i + 1$，$y =$ `a[i]`。下面的几幅图分别对应了下面这些算法的整个排序执行过程的中点。

算法：选择排序、插入排序、归并排序、快速排序、堆排序、基数排序。

请分别将图 13.21 中各图块与对应排序算法对号入坐。

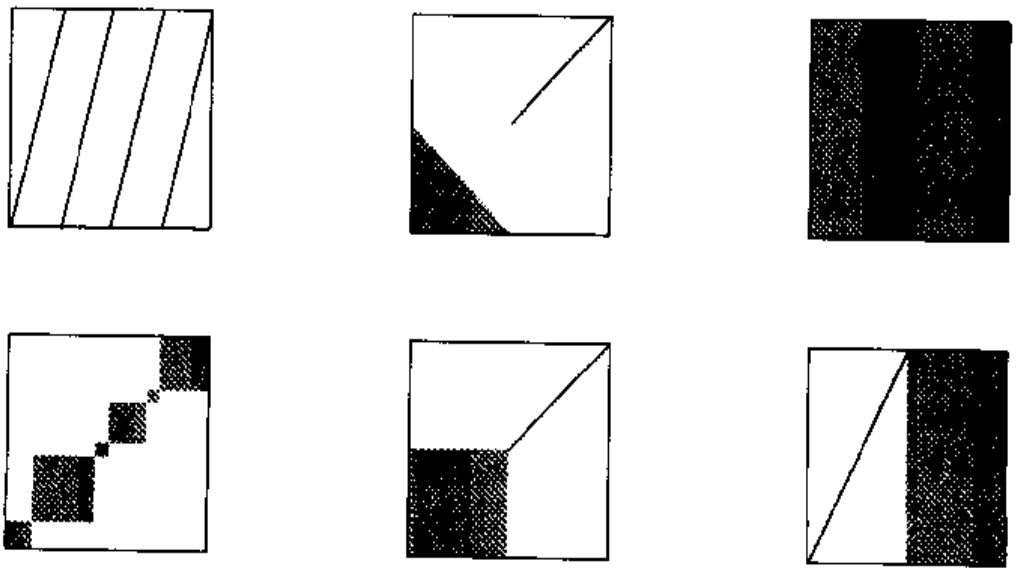


图 13.21 习题 13.19 图

标准分享网 www.bzfxw.com 免费下载

习 题

13.1 如果一个 $O(n^2)$的算法（如冒泡排序、选择排序或插入排序）对一个大小为 200 的数组的排序过程用了 3.1 毫秒，那么，

a. 对于 400 个元素的数组的排序过程要花费多少时间？

b. 对于 40 000 个元素的数组的排序过程要花费多少时间？

13.2 如果一个 $O(n \lg n)$的算法（如归并排序、快速排序或堆排序）对一个大小为 200 的数组的排序过程用了 3.1 毫秒，那么对于 40 000 个元素的类似数组的排序过程要花费多少时间？

13.3 对于一个已经排序的数组，插入排序的时间复杂性是线性的。对于一个逆排序的数组，其时间复杂性是多少？

13.4 冒泡排序在下列条件下的执行情况如何？

a. 对一个已排序的数组进行排序。

b. 对一个逆排序的数组进行排序。

13.5 选择排序在下列条件下的执行情况如何？

a. 对一个已排序的数组进行排序。

b. 对一个逆排序的数组进行排序。

13.6 归并排序在下列条件下的执行情况如何？

a. 对一个已排序的数组进行排序。

b. 对一个逆排序的数组进行排序。

13.7 快速排序在下列条件下的执行情况如何？

a. 对一个已排序的数组进行排序。

b. 对一个逆排序的数组进行排序。

13.8 堆排序在下列条件下的执行情况如何？

a. 对一个已排序的数组进行排序。

b. 对一个逆排序的数组进行排序。

13.9 冒泡排序、选择排序和插入排序都是复杂度为 $O(n^2)$的算法，它们中间哪一个算法最快、哪一个算法最慢？

13.10 归并排序、快速排序和堆排序都是复杂度为 $O(n \lg n)$的算法，它们中间哪一个算法最快、哪一个算法最慢？

13.11 依次用下面 6 种排序算法对数组。

```
inta[] = {44, 77, 55, 99, 66, 33, 22, 88, 77}
```

进行排序，分别手动跟踪它们的排序过程。

a. 冒泡排序。

b. 选择排序。

c. 插入排序。

d. 归并排序。

e. 快速排序。

f. 堆排序。

13.12 对冒泡排序进行修改以使其按照非递增方式对数组进行排序。

13.13 对冒泡排序进行修改以使其具备一定的“智能”：能够在数组一旦被排序好就立即终止。

13.14 证明定理 13.1。

13.15 证明定理 13.2。

13.16 摇动排序（*shaker sort*）类似于冒泡排序，只不过它交替地向上、下两个方向“冒泡”。请实现摇动排序，并确定它是否比插入排序更高效？

13.17 对选择排序（算法 13.2）进行修改以使它在算法第 2 步时使用$\{s_i..s_{n-1}\}$中的最小元素。

13.18 请实现递归版本的选择排序算法。

13.19 证明定理 13.3。

13.20 证明定理 13.4。

13.21 修改插入排序以使它能够间接排序。使用一个独立的索引数组来实现间接性，索引数组的数组元素是原数组的索引。间接排序对索引数组进行操作，而不修改原数组。

13.22 请实现递归版本的插入排序算法。

13.23 证明定理 13.5。

13.24 证明定理 13.6。

13.25 证明定理 13.7。

13.26 修改快速排序以使它不选择子序列的第一个元素，而选择最后一个元素作为基准元素。

13.27 修改快速排序以使它选择第一个元素、中间元素和最后一个元素三者的平均值作为基准元素。

13.28 修改快速排序以使它在数组大小小于 8 时自动切换为插入排序。

13.29 堆排序的时间复杂性为 $O(n \lg n)$，请说明它为什么不总是比最坏情况下复杂性为 $O(n_2)$ 的快速排序算法更好？

13.30 堆排序的时间复杂性为 $O(n \lg n)$，又不需要额外的数组空间，请说明它为什么不总是比需要额外数组空间的归并排序更好？

13.31 证明定理 13.15。

13.32 下面是用拉斯维加斯排序对一叠扑克牌进行排序的描述：

1. 随机洗牌。

2. 如果扑克牌没有排序，重复步骤 1。

请推导这一排序算法的时间复杂性。

标准分享网 www.bzfxw.com 免费下载

复习题答案

13.1 冒泡排序执行速度慢的原因是因为它只是对数据进行局部性操作。各元素每次只移动一个位置。比如，在例 13.1 中，算法一共通过 6 次调用 swap()方法才把元素 99 移动到其正确位置，即 a[8]。

13.2 从 $n(n-1)/2$ 到 $O(n^2)$的推导是基于以下两个事实完成的：

a. 当 n 比较大时（例如 $n>1\,000$），$n(n-1)/2$ 近似于 $n^2/2$。

b. 数据复杂性函数只是用于比较目的的。比如，当对于长一倍的数组进行排序时，需要多用多少时间？对于类似的比较分析，呈线性比的复杂性函数都被认为是等价的。由于 $n^2/2$ 正比于 n^2，系数 1/2 可以忽略，因此得到结论 $O(n^2)$。

13.3 $O(n)$的排序算法（如基数排序和吊桶排序）比 $O(n \lg n)$的排序算法（如归并排序、快速排序和堆排序）速度慢的原因在于，它们的时间复杂性都正比于 n，只是由于额外开销不同使得比例系数不同。而在基数排序和吊桶排序的执行中，每一个循环过程都必须对数组元素进行拷贝操作，因此一般而言它们比较慢。

13.4 归并排序、快速排序和吊桶排序都使用了分治法。

13.5 冒泡排序、选择排序、插入排序、归并排序和快速排序对于链表的排序能够达到对于数组的排序的执行效率。

13.6 快速排序和吊桶排序在最坏情况下比平均情况下慢很多。

13.7 插入排序、SHELL 排序和基数排序在最好情况下比平均情况下快得多。

13.8 因为递归版本在递归调用方法时通常要花费很多开销。

13.9 快速排序使用了分治法：首先它对序列进行 $O(\lg n)$次拆分，然后分别递归地对拆分后的部分进行独立排序，归并排序也使用了分治法，但是在具体做法上的顺序不同：首先进行递归调用，然后再进行 $O(\lg n)$次合并。这两个算法都是 $O(\lg n)$次完成了工作量为 $O(n)$的工作，因此，复杂度为 $O(n \lg n)$。

13.10 当被排序对象是链表或外部文件时，归并排序比另外两个复杂度为 $O(n \lg n)$的算法好。

13.11 在被排序数组已经被排序的条件下，快速排序将退化成选择排序。

13.12 在对大型原始类型的数组进行排序时，快速排序比另外两个复杂度为 $O(n \lg n)$的算法好。

13.13 选择排序可以看成是一种“边输出边排序”的过程：将元素简单地插入数组后，每次选择最大的元素输出。插入排序可以看成是一种“边输入边排序”的过程：在向数组中插入元素时就把它们插入到正确的位置上，输出时只需按照下标顺序输出即可。因此，选择排序的插入过程是 $O(n)$的，输出过程是 $O(n^2)$的。插入排序的插入过程是 $O(n^2)$的，输出过程是 $O(n)$的。总体上，它们都是 $O(n^2)$的。

堆排序可以看成是一种“半个边输出边排序加半个边输入边排序”的过程：在插入

的过程中维护元素部分排序的堆性质，输出时不断地输出第一个元素（最小者）后再维护堆性质。插入和输出都是$O(n \lg n)$的，因此整个算法是$O(n \lg n)$的。

13.14 Java的API使用的是归并排序算法来实现其java.utils.Arrays.sort()方法的。

13.15 SHELL排序、快速排序和堆排序不是稳定的算法。

13.16 归并排序、基数排序和吊桶排序需要额外的数组空间。

13.17 冒泡排序、选择排序、插入排序、归并排序和快速排序特别适合于对以外部文件形式存在的记录进行排序。

13.18 SHELL排序、归并排序、快速排序和吊桶排序是可并行的。

13.19 图块与排序算法的对应如图13.22所示。

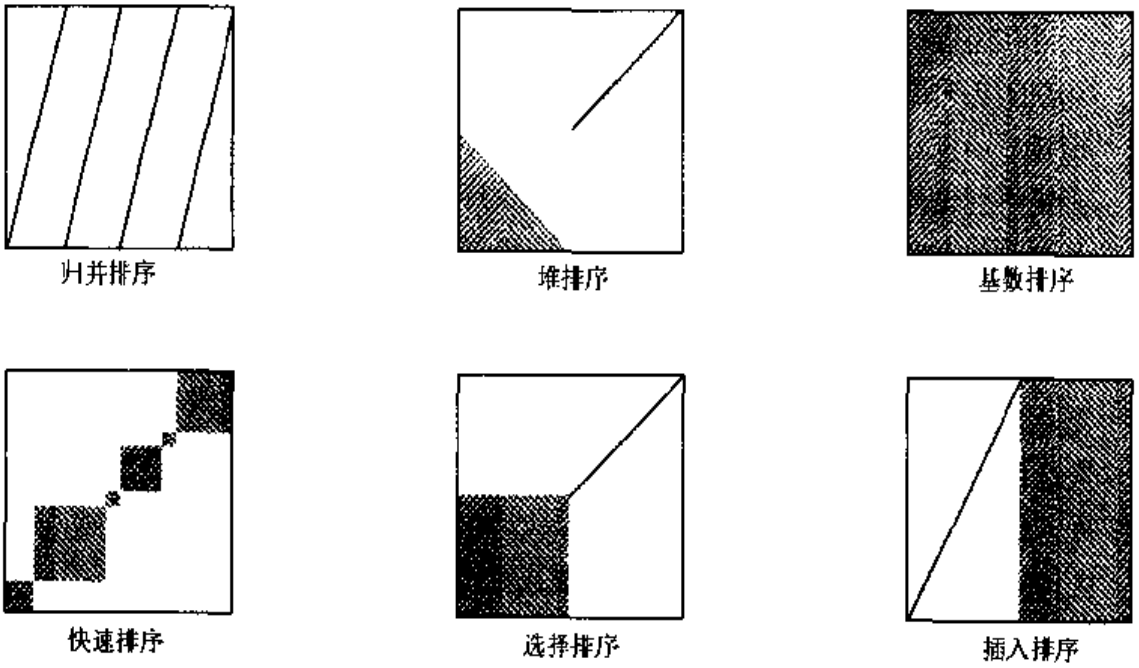


图13.22 习题13.19答案

习 题 解 答

13.1 $O(n^2)$的算法：

a. 对于400个元素的数组的排序过程要花费12.4毫秒（是原来的4倍）。

b. 对于40 000个元素的数组的排序过程要花费124秒（是原来的40 000倍），大约2分钟的时间。这一答案是这样得到的：运行时间t正比于n^2，则存在常量c使得$t = c\,n^2$。如果对n=200个元素排序花了t=3.1毫秒，则有（3.1毫秒）=c（200个元素）^{2}，因此，c=（3.1毫秒）/（200个元素）^{2}=0.000 077 5毫秒/元素^2。因此，对于n=40 000，$t = c\,n^2$ = (0.000 077 5毫秒/元素^2)(40 000元素^2) = 124 000毫秒 = 124秒。

13.2 $O(n \lg n)$的算法对于40 000个元素（400倍长）的数组的排序过程要花费1.24秒。这一答案是这样得到的：运行时间t正比于$n \lg n$，则存在常量c使得$t = c\,n\lg n$。如果

标准分享网 www.bzfxw.com 免费下载

对 n=200 个元素排序花了 t=3.1 毫秒，则有（3.1）$=c$ (200) lg(200)，因此，c=（3.1）/（200）lg（200）=0.015 5/lg(200)。因此，对于 n=40 000，$t=c\,n\lg n$ = (0.015 5/lg(200)) (40 000 lg(40 000)) = 620 (lg(40 000)/lg(lg200)) = 1 240 毫秒 = 1.24 秒。

13.3 对于一个逆排序的数组，插入排序的性能最差. 因为每个元素的插入都会使其左边的所有元素右移一个位置。

13.4 算法 13.1 中实现的冒泡排序是输入不敏感的，也就是说，无论输入的数组是否被排序，是否排逆序，它都要进行 $n(n-1)/2$ 次比较，因此它是低效的。

13.5 选择排序也是输入不敏感的，无论输入的数组是否被排序、是否排逆序，它的运行时间都是基本相同的。

13.6 归并排序也是输入不敏感的，无论输入的数组是否被排序、是否排逆序，它的运行时间都是基本相同的。

13.7 快速排序对输入非常敏感，按照算法 13.6 实现的快速排序算法，如果输入的数组已经被排序或者已经被排逆序，它都将退化成 $O(n^2)$的算法。因为这两种情况下，基准元素将分别被选定为子数组中的两个极值，这就使得对子数组的拆分是极不平衡的，而且需要拆分 n 次，而不是我们所期望的 $\lg n$ 次。

13.8 堆排序对输入有一些敏感性，程度不大。`heapify()`方法一般少于 $\lg n$ 次循环。

13.9 冒泡排序比选择排序慢，插入排序（在大多数情况下）都比它们快。

13.10 归并排序比堆排序慢，快速排序（在大多数情况下）都比它们快。

13.11 图 13.23 给出了 6 种排序跟踪图。

a. 冒泡排序的跟踪图如下：

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]
44	77	55	99	66	33	22	88	77
	55	77						
			66	99				
				33	99			
					22	99		
						88	99	
							77	99
		66	77					
			33	77				
				22	77			
						77	88	
		33	66					
			22	66				
	33	55						
33	44							
	22	44						
22	33							

b. 选择排序的跟踪图如下：

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]
44	77	55	99	66	33	22	88	77
22						44		
	33				77			
		44				55		
			55			99		
						77		99

c. 插入排序的跟踪图如下：

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]
44	77	55	99	66	33	22	88	77
	55	77				44		
		66	77	99				
33	44	55	66	77	99			
22	33	44	55	66	77	99		
						88	99	
						77	88	99

d. 归并排序的跟踪图如下：

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]
44	77	55	99	66	33	22	88	77
44	55	77	99					
				33	66			
							77	88
				22	33	66	77	88
22	33	44	55	66	77	77	88	99

e. 快速排序的跟踪图如下：

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]
44	77	55	99	66	33	22	88	77
			22			99		
				33		77	.	99
22			44					
		44	55					

f. 堆排序的跟踪图如下：

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]
44	77	55	99	66	33	22	88	77
	99		77					
			88				77	
99	44							
	88		44					
			77				44	
77								99
88	77							
44							88	
77	44							
	77		44					
22						77		
77	22							
	66			22				
33					77			
66	33							
	44		33					
22				66				
55		22						
		33	22					
22			55					
44	22							
33		44						
22	33							

图 13.23 习题 13.11 答案

13.12 为了使其按照非递增方式对数组进行排序，对冒泡排序的进行修改如下：

```
public static void sort(int[] a)
{ for (int i=a.length-1; i>0; i--)
```

标准分享网 www.bzfxw.com 免费下载

```
    for (int j=1; j<=i; j++)
      if (a[j-1]<a[j]) swap(a,j-1,j);
  }
```

13.13 “智能”版本的冒泡排序如下：

```
public static void sort(int[] a)
{ boolean sorted=false;
  for (int i=a.length-1; i>0; i--)
  { for (int j=1; j<-i; j++)
    { sorted = true;
      if (a[j-1] > a[j])
      { swap(a,j-1,j);
        sorted = false;
      }
    }
    if (sorted) return;
  }
}
```

13.14 冒泡排序的正确性证明如下：

我们可以根据循环不变量来证明冒泡排序的确对数组进行了排序。在主循环的第1次循环完成后，数组中的最大元素一定被移到了数组的最后一个位置上。因为不管它一开始处在什么位置，它总是一步一步地被移到数组的最后，因为在每次比较时，大的元素总是被后移。正是由于同样的原因，在主循环的第2次循环完成后，倒数第二大的元素将最终被移到数组的倒数第二个位置上。这样，数组中的两个最大的元素都已经被移到正确的位置上了。依此类推，在主循环的每次迭代完成后，循环不变量都为真。在进行了n-1次循环后，n-1个最大的元素都应该处在其正确位置上，当然，第n大元素（也就是数组中的最小元素）此时也只能处在它的正确位置上了，因此，整个数组就被排序了。

13.15 证明冒泡排序的时间复杂性是$O(n^2)$的：

时间复杂性函数$O(n^2)$表示这样一个含义：当n非常大时，循环的迭代次数将正比于n^2。这就是说，如果一个大数组的大小是另外一个数组的2倍，那么对它进行排序的时间将是对另外一个数组进行排序的时间的4倍左右。在外循环的第1次迭代中，内循环将迭代n-1次，在外循环的第2次迭代中，内循环将迭代n-2次，在外循环的第3次迭代中，内循环将迭代n-3次，依此类推。例如，当n=7时，在外循环的第1次迭代中，内循环将迭代6次，在外循环的第2次迭代中，内循环将迭代5次，在外循环的第3次迭代中，内循环将迭代4次，依此类推。总的比较次数为6+5+4+3+2+1=21次。一般情况下，总的比较次数是$(n-1)+(n-2)+(n-3)+\ldots+3+2+1=n(n-1)/2$次（参见定理B.8）。对于较大的$n$来说，它近似等于$n^2/2$，而$n^2/2$又正比于$n^2$。

13.16 摇动排序的实现如下：

```
public static void sort(int[] a)
{ for (int i=a.length; i>0; i -= 2)
  { for (int j=1; j<i; j++)
      if (a[j-1] > a[j]) swap(a,j-1,j);
    for (int j=i-2; j>0; j--)
      if (a[j-1] > a[j]) swap(a,j-1,j);
  }
}
```

13.17 使用{*si*...*sn*-1}中的最小元素的选择排序的修改版本如下：

```
public static void sort(int[] a)
{ for (int i=0; i<a.length-1; i++)
  { int j=i;
    for (int k=i+1; k<a.length; k++)
      if (a[k] < a[j]) j = k;
    swap(a,i,j);    }
}
```

13.18 递归版本的选择排序实现如下：

```
public static void sort(int[] a)
{ sort(a,a.length);
}

public static void sort(int[] a, int n)
{ if (n<2) return;
  int j=0;
  for (int k=1; k<n; k++)
    if (a[k] > a[j]) j = k;
  swap(a,n-1,j);
  sort(a,n-1);
```

13.19 选择排序的正确性证明如下：

最后一个循环不变量证明了算法的正确性。因此，类似于冒泡排序的证明过程，我们只需验证循环不变量的正确性。

在主循环（步骤1）的第1次迭代中，a[i]是数组中的最后一个元素，因而内循环的循环变量k将跑遍a[0]之后的所有数组元素。下标值j的初始值为0，并且每当内循环找到一个更大的元素，j就被赋予该元素的下标。因此，当内循环完成后，a[j]将是数组中的最大元素。这就验证了第一个循环不变量。在外循环后续的每一次迭代中，下标k将跑遍所有未排序的数组部分，依据前面所述的原理，每当内循环结束时，a[j]都是的确为排序数组部分中的最大者。这就验证了，当外循环的每次迭代结束时，第一个循环不变量都是成立的。

由于swap(a,i,j)方法仅仅是交换元素a[i]和a[j]，第二个循环不变量也

标准分享网 www.bzfxw.com 免费下载

就自然成立了。

在第二个循环不变量成立的条件下，根据数学归纳法可得出第三个循环不变量也是成立的。在主循环的第 1 次迭代中，内循环确定 a[j]为数组中的最大元素。swap(a,i,j)方法将最大元素放到数组的最后一个位置 a[i]上，因此 a[i]将大于等于所有 a[j]。在主循环的第 i 次迭代之前，根据归纳假设，有子数组 a[i+1..n-1]已被排序，并且子数组 a[0..i]中的所有元素都小于元素 a[i+1]。在第 i 次迭代后，a[i]将是已被排序的元素中的最小者，即 a[i]<=a[i+1]<=...<=a[n-1]。

13.20 证明选择排序的时间复杂性是 $O(n^2)$的：

同理，证明过程与冒泡排序的时间复杂性证明相类似。在外循环的第 1 次迭代中，内循环将迭代 n–1 次，在外循环的第 2 次迭代中，内循环将迭代 n–2 次，依此类推，最终的总和为$(n-1)+(n-2)+\ldots+2+1=n(n-1)/2$ 次。

13.21 下面的程序段实现并简单测试了间接插入排序：

```
public static void main(String[] args)
{ int[] a = { 77, 44, 99, 66, 33, 55, 88, 22 };
  int[] index = {0,1,2,3,4,5,6,7};
  print(a,index);
  sort(a,index);
  print(a,index);
}

private static void print(int[] a, int[] index)
{ for (int i=0; i<a.length; i++)
    System.out.print(a[index[i]]+" ");
  System.out.println();
}

public static void sort(int[] a, int[] index)
{ for (int i=1; i<a.length; i++)
  { int temp=index[i], j;
    for (j=i; j>0 && a[index[j-1]]>temp; j--)
      index[j] = index[j-1];
    index[j] = temp;
  }
}
```

13.22 递归版本的插入排序实现如下：

```
public static void sort(int[] a)
{ sort(a,a.length);
}
public static void sort(int[] a, int n)
{ if (n<2) return;
```

```
  sort(a,n-1);
  int temp=a[n-1], j;
  for (j=n-1; j>0 && a[j-1]>temp; j--)
    a[j] = a[j-1];
  a[j] = temp;
}
```

13.23 插入排序的正确性证明如下：

在主循环的第 1 次迭代中，a[1]同 a[0]进行比较，如果需要交换，则交换它们。因此，第 1 次迭代结束后，有 a[0] <= a[1]。如果我们假设对于特定的 k，循环不变量在第 k 次循环之前都为真，那么在第 k 次循环完成之后，循环不变量依然将为真，因为在这一次循环过程中，元素 a[k]被插入到小于等于它的元素与大于它的元素之间。根据数学归纳法，循环不变量对所有 k 值都为真。

13.24 证明插入排序的时间复杂性是 $O(n^2)$的：

证明过程与冒泡排序及选择排序的时间复杂性证明相类似。在外循环的第 1 次迭代中，内循环将迭代 1 次，在外循环的第 2 次迭代中，随 a[1]与 a[2]的大小不同，内循环将迭代 1 次或 2 次，在外循环的第 3 次迭代中，内循环将最多迭代 3 次，具体迭代次数取决于元素 a[3]左边共有多少个比它大的元素。依此类推，在外循环的第 k 次迭代中，内循环将最多迭代 k 次。因此，最终的迭代总数为：$1+2+3+...+(n-1)=n(n-1)/2$。

13.25 证明当序列已经被排序时，插入排序的时间复杂性为 $O(n)$的：

在这种情况下，对于外循环的每次迭代，内循环都只需要迭代一次，因此，内循环的总迭代次数为 $1+1+1...+1=n-1$。

13.26 下面的快速排序程序将选择最后一个元素作为基准元素：

```
public static void sort(int[] a)
{ if (a.length>1) sort(a,0,a.length);
}

public static void sort(int[] a, int k, int n)
{ if (n<2) return;
  int pivot = a[k+n-1];
  int i = k-1;
  int j = k+n-1;
  while (i < j)
  { while (a[++i]<pivot) ;
    while (j>0 && a[--j]>pivot) ;
    if (i < j) swap(a,i,j);
  }
  swap(a,i,k+n-1);
  sort(a,k,i-k);
```

标准分享网 www.bzfxw.com 免费下载

```
  sort(a,i+1,k+n-i-1);
 }
```

13.27 下面的快速排序程序段将选择第一个元素、中间元素和最后一个元素三者的平均值作为基准元素：

```
private static void setMedian(int[] a, int i, int j, int k)
{ // POSTCONDITION: either a[i] <= a[j] <= a[k]
  //                 or a[k] <= a[j] <= a[i]
  //  i.e., a[j] is the median of {a[i],a[j],a[k]}
  if (a[i] <= a[k] && a[k] <= a[j]) swap(a,j,k);
  else if (a[j] <= a[i] && a[i] <= a[k]) swap(a,j,i);
  else if (a[j] <= a[k] && a[k] <= a[i]) swap(a,j,k);
  else if (a[k] <= a[i] && a[i] <= a[j]) swap(a,j,i);
}

public static void sort(int[] a)
{ if (a.length>1) sort(a,0,a.length);
}

public static void sort(int[] a, int k, int n)
{ // PRECONDITIONS: 0 <= k <= k+n <= a.length;
  if (n<2) return;
  setMedian(a,k+n/2,k,k+n-1);  // a[k] = median
  int pivot = a[k];
  int i = k;
  int j = k+n;
  while (i < j)
  { while (i+1<k+n && a[++i]<pivot) ;
    while (a[--j]>pivot) ;
    if (i < j) swap(a,i,j);
  }
  swap(a,k,j);
  sort(a,k,j-k);
  sort(a,j+1,k+n-j-1);
}
```

13.28 下面的快速排序程序段在数组大小小于8时自动切换为插入排序：

```
public static void sort(int[] a)
{  if (a.length>1) sort(a,0,a.length);
}

public static void sort(int[]) a,int k,int n)
{  // PRECONDITIONS: 0<= K <= k+n <= a.length;
   if (n<2) return;
```

```
if (n<8)
{  insertionSort(a,k,n);
   return;
}
int pivot = a[k];
int i = k;
int j = k+n;
while (i < j)
{ while (i+1<k+n  && a[++i] <pivot);
 while (a[--j]>pivot);
 if (i<j) swap(a,i,j);
}
swap(a,k,j);
sort(a,k,j-k);
sort(a,j+1,k+n-j-1);
}

public static void insertionSort(int[] a,int k, int n)
{  // POSTCONDITION: a[k] <= a[k+1] <= …<= a[n-1];
   for (int i=k+1; i<n; i++)
   {  int temp=a[i], j;
      for (j=i; j>k && a[j-1] >temp; j--)
        a[j] = a[j-1];
      a[j] = temp;
   }
}
```

13.29 堆排序不总是比快速排序算法更高效的原因是因为它在平均情况下效率并不高。

13.30 堆排序不总是比归并排序算法更高效的原因是因为它是不稳定排序算法。

13.31 堆排序的正确性证明如下：

Heapify 方法的后置条件确保了步骤 3 中的循环不变量的正确性。这也保证了 s_0 是该子序列中的最大元素。步骤 5 将该最大值插入到子序列的最后。因此，当步骤 4 的循环结束时，序列将被排序。Heapify 算法通过在根节点上调用 heapifyDown() 方法来维护堆性质。

13.32 拉斯维加斯排序算法的时间复杂性为 $O(n^n)$。

对于 n 张扑克牌，共存在 $n!$种排列方法。对这些扑克牌的洗牌等价于从这 $n!$种排列中随机地选取一种。而这其中只有一种排列是正确的，也就是顺序的。因此，通过洗牌来排序的洗牌次数应该是 $n!$。对于每一种排列来说，都必须经过 n-1 次比较才能判断该排列是否是顺序的，因此，总的复杂性为 $O(n\ n!)$。根据 Stirling 公式（参见附录 A.2），$O(n\ n!)=O(n^n)$。

标准分享网 www.bzfxw.com 免费下载

第14章 表

表（也称作映射、待查表、关联数组或者字典）是一种容器型的数据结构，它允许通过任意索引类型对其中元素的直接访问。它和数组或向量有些相似，只不过索引变量不必一定是整数类型。这种数据结构的一个很好的类比就是字典，如图 14.1 所示：索引变量就是要查的单词，而被这个索引变量索引的元素则是这个单词的所有注解部分。

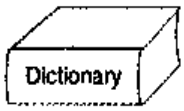


图 14.1 字典

表是序偶的序列。序偶的第一部分称为键（*key*），它是表中的索引，功能类似于数组中的整数下标。序偶的第二部分称为对应键的值。它包含了被查找的信息。在字典的例子中，键就是待查的单词，值就是字典中该单词的注解（以及出现在注解位置的其他的内容）。

表也称为映射（map），这是因为我们可以这样想：表的键被映射到它的值，就像数学函数 *f*(*key*) = *value*。表也称为关联数组（association array），这是因为它们可以用两个并列的数组加以实现：键处在一个数组中，值处在另外一个数组中。

14.1 Java 的 Map 接口

java.util 包中的 Map 接口的定义如下（参见图 14.2）：

```
public interface Map
{  int size();
boolean isEmpty();
boolean containsKey(Object key);
boolean containsValue(Object value);
Object get(Object key);
Object put(Object key,Object value);
Object remove(Object key);
Void putAll(Map map);
Void clear();
Public Set keySet();
Public Collection values();
Public Set entrySef();
Public interface Entry
{  Object getKey();
Object getValue();
Object setValue(Object value);
```

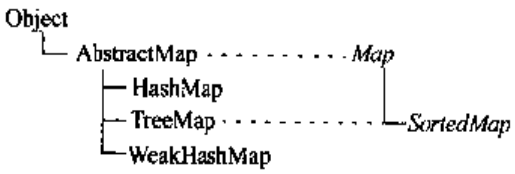


图 14.2 Map 接口

```
boolean equals(Object o);
int hashCode();
}
boolean equals(Object o);
int hashCode();
}
```

14.2 HashMap类

如前面的类层次图所示，Java 为 Map 接口定义了 4 个实现：类 AbstractMap，类 HashMap、类 TreeMap 和类 WeakHashMap。

例 14.1 一个德英字典

下面这段程序使用类 HashMap 来构造了一个德英字典：

```
import java.util.*;

public class Ex1401
{ public static void main(String[] args)
  { Map map = new HashMap();
    map.put("Tag","day");
    map.put("Hut","hat");
    map.put("Uhr","clock");
    map.put("Rad","wheel");
    map.put("Ohr","ear");
    map.put("Tor","gate");
    System.out.println("map=" + map);
    System.out.println("map.size="+map.size())
    System.out.println("map.keySet()=" + map.keySet());
    System.out.println("map.values()=" + map.values());
    System.out.println("map.get(\"Uhr\")=" + map.get("Uhr"));
    System.out.println("map.remove(\"Rad\")=" + map.remove("Rad"));
    System.out.println("map.get(\"Rad\")=" + map.get("Rad"));
    System.out.println("map=" + map);
    System.out.println("map.size="+map.size())
  }
}
```

上述程序的输出为：

```
map={Rad=wheel, Uhr=clock, Ohr=ear, Tor=gate, Hut=hat, Tag=day}
map.size()=6
map.keySet()=[Rad, Uhr, Ohr, Tor, Hut, Tag]
map.values()=[wheel, clock, ear, gate, hat, day]
```

标准分享网 www.bzfxw.com 免费下载

```
map.get("Uhr")=clock
map.remove("Rad")=wheel
map.get("Rad")=null
map={Uhr=clock, Ohr=ear, Tor=gate, Hut=hat, Tag=day}
map={Uhr=clock,Ohr=ear,Tor=gate,Hut=hat,Tag=day
map.size()=5
```

上面，put()方法将键/值对儿插入到表中，例如：

```
map.put("Tag", "day");
```

将键/值对（"Tag","day"）插入到字典中，其中“Tag”是键，“day”是值。

第一次对 println()方法的调用将引发 HashMap.toString()方法，打印出整个 map 对象。第二次对 println()方法的调用将引发 HashMap.size()方法，表明这个 map 对象包括 6 个键/值对。接下来对 println()方法的调用引发了 HashMap.keySet()方法，方法返回了一个 Set 对象，它包含了所有键（即 6 个德语单词）。接下来对 println()方法的调用引发了 HashMap.values()方法，方法返回了一个 Collection 对象，它包含了所有值（即 6 个英语单词）。接下来对 println()方法的调用引发了 HashMap.get()方法，方法返回了给定键对应的值，这次调用返回了键对象“Uhr”对应的值对象“clock”。接下来对 println()方法的调用引发了 HashMap.remove()方法，方法删除了键/值对（“Rad”，“wheel”）。接下来的方法调用证实了上一次删除操作已经成功完成，因为方法调用 map.get("Rad")返回了一个 null 对象。最后两个 println()语句再一次打印了 map 对象的内容和大小，表明了（“Rad”，“wheel”）确实已经被删除。

例 14.1 中 HashMap 对象中各键/值对的存储顺序似乎是随机的，与插入的顺序无关。下一个例子中，我们将再次验证这一点。

例 14.2 Java 的 HashMap 对象是 Hash 表

下面的程序创建两个独立的 HashMap 对象，并用同样的键/值填充它们，但是填充的顺序不同：

```
import java.util.*;

public class Ex1402
{ public static void main(String[] args)
  { Map map1 = new HashMap();
    map1.put("Tor","gate");
    map1.put("Rad","wheel");
    map1.put("Tag","day");
    map1.put("Uhr","clock");
    map1.put("Hut","hat");
```

```
        map1.put("Ohr","ear");
        System.out.println("map1=" + map1);
        Map map2 = new HashMap();
        map2.put("Rad","wheel");
        map2.put("Uhr","clock");
        map2.put("Ohr","ear");
        map2.put("Tag","day");
        map2.put("Tor","gate");
        map2.put("Hut","hat");
        System.out.println("map2=" + map2);
    }
}
```

上述程序的输出为：

```
map1={Rad=wheel, Uhr=clock, Ohr=ear, Tor=gate, Hut=hat, Tag=day}
map2={Rad=wheel, Uhr=clock, Ohr=ear, Tor=gate, Hut=hat, Tag=day}
```

从 toString()方法的输出可以看出 HashMap 表中的键/值的存储顺序。两个表中的存储顺序是相同的，与它们的插入顺序无关。可以看到，这一顺序与例 14.1 中的顺序是相同的。

14.3 Java 的 Hash 代码

HashMap 表中键/值对的存储顺序取决于表的容量和这些对象的值。在 3.4 节中曾经介绍过，Java 中每个对象都有一个固有的 hash 代码，它是通过存储在该对象中的实际数据计算出来的。Object.hashCode()方法返回每个对象的 hash 代码。

例 14.3 某些 String 对象的 Hash 代码

下面的代码将前面例子中的各 String 对象的 hash 代码打印出来：

```
import java.util.*;

public class Ex1403
{ public static void main(String[] args)
  { printHashCode("Rad");
    printHashCode("Uhr");
    printHashCode("Ohr");
    printHashCode("Tor");
    printHashCode("Hut");
    printHashCode("Tag");
  }

  private static void printHashCode(String word)
```

标准分享网 www.bzfxw.com 免费下载

```
  { System.out.println(word+": "+word.hashCode());
  }
}
```

上述程序的输出为：

```
Rad: 81909
Uhr: 85023
Ohr: 79257
Tor: 84279
Hut: 72935
Tag: 83834
```

所有 6 个代码都是 5 位数的整数，这表明所有这些 String 对象的长度都为 3。

14.4 Hash 表

Hash 表是表的一种类型，它使用一个特别的函数从键值计算出数据值的存储位置，而不将键值本身存储在表中。这个特别的函数称为该表的 Hash 函数。由于查找时间与表的大小无关，因此 Hash 表的存取时间是非常快的。

Java 在它的 java.util 包中定义了 HashTable 类。但是它已经被升级为 HashMap 类。也就是说，HashMap 表能够完成 HashTable 表能完成的所有功能。除此之外，HashMap 类与 Java Collections 框架中的其余部分更加一致。

一般的 Hash 表看起来如图 14.3 所示：一个通过 hash 值索引的对象数组。这就要求 Hash 函数的值域必须同数组的索引变量的取值范围相匹配，在实现中，可以简单地通过使用求余运算符来实现。



图 14.3 Hash 表的存储结构

例 14.4 将键映射到大小为 11 的 Hash 表

下面这段程序将存储在大小为 11 的 Hash 表中的 String 类型的对象的 Hash 代码打印出来：

```
import java.util.*;

public class Ex1404
{ private static final int MASK = 0x7FFFFFFF; // 2^32-1
  private static final int CAPACITY = 11;

  public static void main(String[] args)
  { printHash("Rad");
    printHash("Uhr");
    printHash("Ohr");
```

```
    printHash("Tor");
    printHash("Hut");
    printHash("Tag");
  }

  private static void printHash(String word)
  { System.out.println("hash(" + word + ") = " + hash(word));
  }

  private static int hash(Object object)
  { return (object.hashCode() & MASK) % CAPACITY;
  }
}
```

上述程序的输出为：

```
hash(Rad) = 3
hash(Uhr) = 4
hash(Ohr) = 2
hash(Tor) = 8
hash(Hut) = 5
hash(Tag) = 3
```

上面的程序段中，Hash 函数值的计算是通过如下语句完成的：

```
return (object.hashCode() & MASK) % CAPACITY;
```

其中，MASK 是 11，CAPACITY 是 2 147 483 647，用十六进制表示为 0x7FFFFFFF。操作 n & MASK 的作用是将整数 n 的符号位去掉。在 Java 中使用求余运算符计算数组的索引之前最好进行这一操作，因为与 C++不同，在 Java 中，如果 m 是负数，则 m % CAPACITY 的结果也是负数。在本例中，方法 hash() 的返回结果必定落在 0 到 10 之间。

前 5 个字符串分别被 Hash 到索引值 3，4，2，8 和 5，因而它们被分别存储在 Hash 表中的相应位置上。但是第 6 个字符串（“Tag”）也被 Hash 到 3，出现了与“Rad”的冲突，因为“Rad”早就存储在 3 的位置上了。当这类冲突出现后，最常用的一种算法就是简单地将这个新加入的元素存放在后续的空位中。在本例中将是位置 6，因为位置“Uhr”已经处在位置 4 中，“Hut”已经处在位置 5 中。这种冲突消解算法称为线性探测算法。

类 HashMap 就是使用了类似于例 14.4 中的 Hash 函数来实现其成员访问方法的，这些方法包括：containsKey()，get()，put()，remove()以及 entrySet()。一开始，Hash 表的大小被设为 101。了解到这些之后，我们就不难知道在前面的例子中，为什么那 6 个字符串是按照特定的顺序存储的了。

例 14.5 将键映射到大小为 101 的 Hash 表

下面的程序段同例 14.4 基本相同，除了 Hash 表的大小 CAPACITY 为 101 而不是 11：

标准分享网 www.bzfxw.com 免费下载

```
import java.util.*;

public class Ex1405
{ private static final int MASK = 0x7FFFFFFF; // 2^32-1
  private static final int CAPACITY = 101;

  public static void main(String[] args)
  { printHash("Rad");
    printHash("Uhr");
    printHash("Ohr");
    printHash("Tor");
    printHash("Hut");
    printHash("Tag");
  }

  private static void printHash(String word)
  { System.out.println("hash(" + word + ") = " + hash(word));
  }

  private static int hash(Object object)
  { return (object.hashCode() & MASK) % CAPACITY;
  }
}
```

上述程序的输出为：

```
hash(Rad) = 99
hash(Uhr) = 82
hash(Ohr) = 73
hash(Tor) = 45
hash(Hut) = 13
hash(Tag) = 4
```

程序的运行结果显示元素的存储次序同它们的访问次序是相反的。

14.5 Hash表的性能

如果用大小为101的Hash表存储6个元素，该Hash表的性能将是非常高的，出现冲突的机会非常小，访问函数的执行也是非常快的：只需 $O(1)$的时间。这就是直接访问，就像是数组一样。

但是如果用大小为101的Hash表来存储100个元素，那么Hash表的性能就会非常糟糕了，因为在存储元素的过程中将不可避免地出现很多冲突情况。例如，如果字符串“Lob”在存取操作时发生了60次冲突才找到一个空位，那么在以后的每一次对它的存取中都必须探测60次才能找到它。这样的工作效率就已经逼近 $O(n)$了，也就是说，并不比链表快多少。

解决这一问题的办法就是避免 Hash 表的过分拥挤，一般在它存储的元素数到达某一阀值时就增大它的容量。

Hash 表的拥挤程度取决于两个参数：表征 Hash 表中实际元素个数的大小（*size*），表征 Hash 表中元素位置的容量（*capacity*）。这两个参数的比例称为负载系数（*load factor*）。在本节的第一个例子中，Hash 表的大小是 6，容量是 101，则负载系数为 6/101＝5.94%。在第二个例子中，Hash 表的大小是 100，则负载系数为 100/101＝99.01%。

类 HashMap 能够在负载系数达到某个阀值时自动增加 Hash 表的容量。阀值可以在 Hash 表创建时通过下面的构造函数来设置：

```
public HashMap(int initialCapacity, float loadFactor)
```

当然，也可以通过上面这个构造函数来设置初始容量。如果使用的构造函数不对这两个参数指定值，则容量的缺省值为 101，负载系数的缺省值为 75%。

14.6 冲突消解算法

前面的例子中使用的冲突消解算法称为线性探测。当一个元素被 Hash 函数定位到一个已经使用的位置上时，算法将递增位置索引直至找到一个空位置。当然，如果到了 Hash 表的最后，就跳回到表的起始处。

例 14.6 线性探测

下面的程序段扩展了例 14.4，它记录了 Hash 表的哪一个表项被使用了，已经每次调用 Hash 函数后的负载系数。

```
import java.util.*;

public class Ex1406
{ private static final int MASK = 0x7FFFFFFF; // 2^32-1
  private static final int CAPACITY = 11;
  private static int size=0;
  private static boolean[] used = new boolean[CAPACITY];

  public static void main(String[] args)
  { printHash("Rad");
    printHash("Uhr");
    printHash("Ohr");
    printHash("Tor");
    printHash("Hut");
    printHash("Tag");
    printHash("Eis");
    printHash("Ast");
```

标准分享网 www.bzfxw.com 免费下载

```
    printHash("Zug");
    printHash("Hof");
    printHash("Mal");
  }

  private static void printHash(String word)
  { System.out.println("hash(" + word + ") = " + hash(word)
          + ", load = " + 100*size/CAPACITY + "%");
  }

  private static int hash(Object object)
  { ++size;
    int h = (object.hashCode() & MASK) % CAPACITY;
    while (used[h])
    { System.out.print(h + ", ");
      h = (h+1)%CAPACITY;
    }
    used[h] = true;
    return h;
  }
}
```

上述程序的输出为：

```
hash(Rad) = 3, load = 9%
hash(Uhr) = 4, load = 18%
hash(Ohr) = 2, load = 27%
hash(Tor) = 8, load = 36%
hash(Hut) = 5, load = 45%
3, 4, 5, hash(Tag) = 6, load = 54%
5, 6, hash(Eis) = 7, load = 63%
3, 4, 5, 6, 7, 8, hash(Ast) = 9, load = 72%
9, hash(Zug) = 10, load = 81%
3, 4, 5, 6, 7, 8, 9, 10, hash(Hof) = 0, load = 90%
2, 3, 4, 5, 6, 7, 8, 9, 10, 0, hash(Mal) = 1, load = 100%
```

其中，size 中包含了表中元素的个数。数组 used[]标识了 Hash 表中哪些已经被使用。printHash()方法打印出 Hash 表的索引以及负载系数。如果需要用到线性探测，则各个被探测到的索引也将被一起打印出来。

在例 14.4 中已经看到，在插入“Tag”时发生了冲突。上面的程序段表明它发生了 3 次冲突（分别在索引 3、4 和 5 处）才在索引 6 处找到了一个空位。在该次插入操作之后，表已经被使用了 54%。

从那以后的每个元素的插入都导致了冲突。当然，随着表越来越满，冲突的频率也就越来越高。最后一个元素“Hut”共发生了 10 次冲突。这也表明，从此以后，每次访问这

个元素时，都必须扫描11个元素才能找到它，这显然已经是$O(n)$时间的了。

另外一个冲突消解算法是平方探测（*quadratic probing*）。一般情况下，它的性能比线性探测要好。这一算法在探测时会跳过很多元素，从而使得使用过的位置比较均匀，而不会聚在一起。性能的提高主要是因为这一算法的探测链比较短。

关于上面的程序段，还需要注意的是在元素“Mal”插入时，索引实际上已经回卷了，即索引依次为：2，3，4，5，6，7，8，9，10，0，1。

例14.7 平方探测

下面的程序段的其他部分与例14.6的相同，除了函数hash()被修改成下面的内容：

```
private static int hash(Object object)
{ ++size;
  int h = (object.hashCode() & MASK) % CAPACITY;
  if (used[h])
  { int h0=h;
    int jump=1;
    while (used[h])
    { System.out.print(h + ", ");
      h = (h0+jump*jump)%CAPACITY;
      ++jump;
    }
  }
  used[h] = true;
  return h;
}
```

上述程序的输出为：

```
hash(Rad) = 3, load = 9%
hash(Uhr) = 4, load = 18%
hash(Ohr) = 2, load = 27%
hash(Tor) = 8, load = 36%
hash(Hut) = 5, load = 45%
3, 4, hash(Tag) = 7, load = 54%
5, hash(Eis) = 6, load = 63%
3, 4, 7, hash(Ast) = 1, load = 72%
hash(Zug) = 9, load = 81%
```

关键的区别在于当冲突发生时，在while循环中计算下一个准备探测的索引。不同于线性探测的线性搜索，这里使用的是平方的增量。例如，当插入“Ast”时，在位置3处产生冲突，线性探测将依次探测4，5，6，7，8和9（参见例14.6）。但是平方探测只探测3，4，7和1（=12 mod 11），分别使用的增量为1，4，9（1^2，2^2，和3^2）。相比之下，线性探测要多花费50%的探测次数。

标准分享网 www.bzfxw.com 免费下载

一方面，平方探测能够提高性能，但是另一方面，在平方探测时可能会出现无限循环。在例 14.7 中，如果继续插入“Hof”就会出现这种情况。字符串“Hof”一开始被定位到位置 3，在经过 8 次冲突后，最终找到了位置 0（=11 mod 11）这个空位。但是这个元素的探测序列与字符串“Ast”的冲突序列是相同的：3，4，7，1，8，6，6，8，1，7，4，3，4……。这是从没有进行模运算的序列：3，4，7，12，19，28，39，52，67，84，103，124，147……中得到的。这一过程将无休止的进行下去，每次探测的位置不外乎 3，4，7，1，8 和 6，而这些位置又都被占用了。因此，在这种情况下，尽管 Hash 表只有 81%的空间被占用，但是插入操作还是要宣告失败。在线性探测算法中，不会发生类似情况。

14.7 独立链

如果不把希望寄托在更高效的冲突消解算法上，实际上还有一种办法可以避免冲突，这就是允许 Hash 表的每个空位放置多个元素。这一方法被称为对立链（*searate chaining*）。由于它使用的是链表（链）来存放多个元素，表的空位又被称作吊桶（buckets）。

例 14.8 独立链

下面的程序段就是使用了独立链的类 HashTable 的部分定义：

```
putblic class HashTable
{  private static final int MASK=0x7FFFFFFF;//2^32-1
   private static int capacity=101;
   private static int size=0
   private static float load=0.75F;
   private static LinkedList [] buckets;

   HashTable()
{  buckets=new LinkedList [capacity];
   for (int i=0;i<capacity;i++)
     buckets[i]=new LinkedList();
}
HashTable (int capacity,float load)
{  this();
   this.capacity=capacity;
   this.load=load;
}
Object put (Object key,Object value);
{  int h=hash(key);
   LinkedList bucket=buckets[h];
   object oldValue=null;
   for (ListIterator it=bucket.iterator();it.hasNext();)
   {  Map.Entry entry=it.next();
```

```
        if (entry.getKey().equals(key))break;
    }
    if(entry.getkey().equals(key))oldvalue=entry.setValue(value);
    else bucket.add(new Entry(key,value));
    return oldValue;
}
// more methods...
}
```

请注意方法put()有两个目的。如果表中已经有了给定键的项，该方法只是更改该键对应的值，如果表中还没有该键，就将键/值对加入到表中。

java.util.HashMap类就是使用的与例14.8相类似的独立链。

14.8 应用

表被广泛应用于系统程序设计。另外，它们在关系数据库中的也是基础数据结构。

下面是一些应用程序设计的例子。

例14.9 字句索引

所谓字句索引（*concordance*）是一篇文档中出现的若干单词的列表，列表中列出这些单词出现的行号。这类似于一本书的目录，不同点是用行号代替页码。字句索引在分析一个文件，找出单词出现频率及其关联时很有用，而通过直接读取文件来获得这些信息是很困难的。

该程序建立了一个文本文件的字句索引。这里程序运行使用的特殊文本文件选自莎士比亚的*Julius Caesar*剧。第一部分的字句索引结果见表14.1。

该输出是通过先获得一个字句索引集合视图，然后在该集合上迭代得到。每个元素是一个映射。Entry对象的关键字是文本文件中的单词（上面的大写单词），值是找到单词的行号。例如，单词man在10，15，22，27行出现。第10行是

```
For Brutus is an honourable man ;
```

下面是完整的程序：

```
import java.io.*;

import java.util.*;
public class Ex1409
{ private static Map concordance;
  private static File file;
  private static FileReader reader;
  private static BufferedReader in;
     public static void main(String[] args)
```

标准分享网 www.bzfxw.com 免费下载

表 14.1 字句索引示例

Shakespeare.txt

```
Friends, Romans, countrymen, lend me your ears!
I come to bury Caesar, not to praise him.
The evil that men do lives after them,
The good is oft interred with their bones;
So let it be with Caesar.  The noble Brutus
Hath told you Caesar was ambitious;
If it were so, it was a grievous fault;
And grievously hath Caesar answer'd it.
Here, under leave of Brutus and the rest, --
For Brutus is an honourable man;
So are they all, all honourable men.
Come I to speak in Caesar's funeral.
He was my friend, faithful and just to me.
But Brutus says he was ambitious;
And Brutus is an honourable man.
He hath brought many captives home to Rome.
Whose ransoms did the general coffers fill:
Did this in Caesar seem ambitious?
When that the poor have cried, Caesar hath wept;
Ambition should be made of sterner stuff.
Yet Brutus says he was ambitious;
And Brutus is an honourable man.
You all did see that on the Lupercal
I thrice presented him with a kingly crown,
Which he did thrice refuse: was this ambition?
Yet Brutus says he was ambitious;
And, sure, is an honourable man.
I speak not to disprove what Brutus spoke,
But here I am to speak what I do know.
You all did love him once, not without cause.
What cause withholds you, then, to mourn for him?
O judgement! thou art fled to brutish beasts,
And men have lost their reason!
```

Ex1409.out

```
STUFF=20
THE=3, 4, 5, 9, 17, 19, 23
GRIEVOUS=7
GRIEVOUSLY=8
WHOSE=17
REASON=33
AND=8, 9, 13, 15, 22, 27, 33
FAULT=7
KINGLY=24
COUNTRYMEN=1
MOURN=31
FRIENDS=1
GOOD=4
LEAVE=9
ROME=16
CROWN=24
SHOULD=20
INTERRED=4
WEPT=19
FOR=10, 31
FRIEND=13
BUT=14, 29
BRUTUS=5, 9, 10, 14, 15, 21, 22,
26, 28
MAN=10, 15, 22, 27
CAUSE=30, 31
SURE=27
PRESENTED=24
YOU=6, 23, 30, 31
SEE=23
BONES=4
LIVES=3
REFUSE=25
HERE=9, 29
```

```
{ new Ex1409("Shakespeare.txt");
  load();
dump();
}
  Ex1409(String filename)
```

```
  { try
    { concordance = new HashMap();
      file = new File(filename);
      reader = new FileReader(file);
      in = new BufferedReader(reader);
    }
    catch(Exception e) { System.out.println(e); }
  }

  private static void load()
  { String line=null;
    StringTokenizer parser=null;
    int lineNumber=0;
    try
    { while ((line=in.readLine()) != null)
      { ++lineNumber;
        parser = new StringTokenizer(line,",.;:()-!?' ");
        while (parser.hasMoreTokens())
        { String word = parser.nextToken().toUpperCase();
          String listing = (String)concordance.get(word);
          if (listing==null) listing = "" + lineNumber;
          else listing += ", " + lineNumber;
          concordance.put(word,listing);
        }
      }
    }
    catch(Exception e) { System.out.println(e); }
  }

  private static void dump()
  { Set set = concordance.entrySet();
    for (Iterator it=set.iterator(); it.hasNext(); )
      System.out.println(it.next());
  }
}
```

main()方法实例化 EX1409 类，将字符串"Shakespear.txt"传递给它的构造函数。构造函数实例化 HashMap 类，创建 concordance 对象。它同时也实例化输入文件的读对象 in。

main()方法调用 load()方法，load()方法从文本文件中装入字句索引。这一方法使用了 StringTokenizer 对象 parser 来分析文本信息，通过方法 nextToken()来抽取各个单词。外层 while 循环的每一次循环读一行信息，内层 while 循环的每一次循环读一个单词。语句

标准分享网 www.bzfxw.com 免费下载

```
String listing = (String) concordance.get(word)
```

在字句索引中查找作为键的单词。如果找到了相应的值被赋予到字符串 listing 中，否则，listing 就为 null。如果为 null，listing 将被赋予一个包含行号的字符串，否则，行号将被追加到已有的 listing 中。无论哪一种情况，语句

```
concordance.put(word, listing)
```

都将新的 listing 加入的 Hash 表中，区别在于是作为一个新表项加入还是作为已有表项的更新加入。

注意到，parser 对象的创建是通过下面这条对构造函数的调用完成的：

```
parser = new StringTokenizer(line, ",.;:()-!? ");
```

它使用的是带两个参数的 StringTokenizer 类的构造函数，第二个参数是由所有作为分隔符的字符组成的字符串。通过将逗号“,”、句号“.”、分号“;”等罗列出来，实际上就可以将它们从存储的单词中剔除出去了。

最后，main()调用 dump()方法来打印字句索引。由于 HashMap 类中没有直接的遍历手段（参见 14.1 节），所以必须先把它复制成一个 Set 对象。这样，我们就必须从另外一个视角来看待 HashMap 对象：把它看成是以表中各表项为元素的集合。然后，由于 Set 接口提供了 iterator()方法，我们就可以通过一个 for 循环很方便地遍历它了。叠代器的 next()方法将表项返回为 Object 对象，这个 Object 对象的固有类型是 Map.Entry（Entry 是 Map 的一个内部接口，参见 14.1 节）。因此，当这一对象被传递给 println()方法时，其 Map.Entry.toString()方法被激活，产生了如下所示的输出信息：

```
BRUTUS=5, 9, 10, 14, 15, 21, 22, 26, 28
```

等号“=”将各表项的键和值分开。在这个例子中，键是“BRUTUS”，它的值是“5，9，10，14，15，21，22，26，28”。

字句索引被用于分析文本信息。从这个例子中可以看出，在莎士比亚的这个戏剧中，更多的内容是关于 BRUTUS 的。它表明有 9 处引用了“Brutus”（多于对“Caesar”的引用），还表明了“Brutus”的同义词表示“Brutish”。

例 14.9 的输出也表明了 Hash 表的一个关键特点：它们的内容没有被排序。为了能够获得一个按照字母序排列的输出形式，必须对内容进行排序操作。

14.9 TreeMap 类

TreeMap 类扩展了 AbstractMap 类，并实现了 SortedMap 接口（参见 14.1 节）。它之所以被称为 TreeMap 是因为它是一棵二叉搜索树，而不是 Hash 表。但是它仍然是一个带有键/值对的映射。作为一种树结构，它失去了高效的 $O(1)$访问时间，但是却换来了能够被排序的能力。

例 14.10 排序的字句索引

下面的程序段与例14.9的基本相同，除了在构造函数中将HashMap类换成了TableMap类：

```
Ex1410(String filename)
{ try
  { concordance = new TreeMap();
    file = new File(filename);
    reader = new FileReader(file);
    in = new BufferedReader(reader);
  }
  catch(Exception e) { System.out.println(e); }
}
```

作为程序运行结果的字句索引被存储成一棵排序的二叉搜索树，因此，其输出结果为：

```
A=7, 24
AFTER=3
ALL=11, 11, 23, 30
AM=29
AMBITION=20, 25
AMBITIOUS=6, 14, 18, 21, 26
AN=10, 15, 22, 27
AND=8, 9, 13, 15, 22, 27, 33
ANSWER=8
ARE=11
ART=32
BE=5, 20
BEASTS=32
BONES=4
BROUGHT=16
BRUTISH=32
BRUTUS=5, 9, 10, 14, 15, 21, 22, 26, 28
BURY=2
BUT=14, 29
CAESAR=2, 5, 6, 8, 12, 18, 19
CAPTIVES=16
CAUSE=30, 31
COFFERS=17
COME=2, 12
COUNTRYMEN=1
CRIED=19
CROWN=24
D=8
DID=17, 18, 23, 25, 30
```

标准分享网 www.bzfxw.com 免费下载

```
DISPROVE=28
DO=3, 29
EARS=1
EVIL=3
FAITHFUL=13
FAULT=7
FILL=17
FLED=32
FOR=10, 31
FRIEND=13
FRIENDS=1
FUNERAL=12
GENERAL=17
GOOD=4
GRIEVOUS=7
GRIEVOUSLY=8
HATH=6, 8, 16, 19
HAVE=19, 33
HE=13, 14, 16, 21, 25, 26
HERE=9, 29
HIM=2, 24, 30, 31
```

除了展示 Hash 表的线性时间、无排序能力，以及二叉搜索树的平方时间、有排序能力外，例 14.10 还展示了 Java Collection 框架的卓越的设计和效率。由于 HashMap 类和 TreeMap 类都实现了同一个 Map 接口，因此我们可以仅仅通过改变它们的构造函数的名字来将一种类替换成另一种类。

复 习 题

14.1 表与向量之间的差别是什么？

14.2 为什么表也称为映射？

14.3 为什么表也称为关联数组？

14.4 为什么表也称为字典？

14.5 什么是字句索引？

14.6 什么是 Hash 表？

14.7 Java 的 Hashtable 类与 HashMap 类之间的差别是什么？

14.8 本章的前两个例子表明，如果不存在冲突，向 Hash 表中插入的顺序是无关的。如果冲突存在，情况又怎么样呢？

14.9 平方探测与线性探测相比，有哪些优缺点？

14.10 使用 HashMap 同使用 TreeMap 相比，有哪些优缺点？

习　题

14.1 运行类似于例 14.1 的程序，并将以下 16 个表项插入到德英字典中：

```
map.put("Ast","gate");
map.put("Eis","ice");
map.put("Hof","court, yard, farm");
map.put("Hut","hat");
map.put("Lob","praise");
map.put("Mal","mark, signal");
map.put("Mut","courage");
map.put("Ohr","ear");
map.put("Ost","east");
map.put("Rad","wheel");
map.put("Rat","advice, counsel");
map.put("Tag","day");
map.put("Tor","gate");
map.put("Uhr","clock");
map.put("Wal","whale");
map.put("Zug","procession, train");
```

14.2 修改 Concordance 类，使得它能够过滤掉一些常用词汇（代词、副词等），对这些常用词汇的统计无助于对文档的分析。将这些常用词汇存储在一个独立的文件中，这个文件的名字为 CommonWords.txt，其内容如图 14.4 所示。

CommonWords.txt

```
A
AFTER
ALL
AM
AN
AND
ARE
BE
BROUGHT
BUT
COME
DID
DO
FOR
HATH
HAVE
HE
HERE
HIM
I
IF
IN
IS
IT
LET
MADE
MANY
O
OF
ON
```

图 14.4 CommonWords.txt

14.3 修改例 14.1 的程序，以使它能够按照字母序来存储单词。让它装入与习题 14.1 相同的数据，然后按照字母序打印表的内容。

14.4 实现一个 FrequencyTable 类，其功能是：统计一个文本文件中单词的频率，输出这些单词及其频率的列表。

复习题答案

14.1 向量通过其整数索引提供对其元素的直接访问。表通过特定的键域提供对其元素的直接访问，该键域可以是任何可排序的类型：int，double，String 等。

14.2 表之所以被称为映射，是因为像数学函数一样，它将每个键映射到一个值。

14.3 表之所以被称为关联数组，是因为它可以像数组一样工作，每个键值都与一个数组元素相关联，同样，像数学函数一样，它将每个键映射到一个值。

14.4 表之所以被称为字典，是因为它可以按照日常生活中的字典一样使用：对元素的查找，

标准分享网 www.bzfxw.com 免费下载

就像是对字典中单词的查找一样。

14.5 所谓字句索引，就是一篇文档中出现的若干单词的列表，列表中列出这些单词在该文档中出现的行号。

14.6 Hash 表是表的一种。它通过特殊的函数由数据的键计算出数据值的存储位置，而不把键存放在表中。

14.7 差别不大。Java 的 Hashtable 类几乎可以被 HashMap 类取代，因为后者与 Java Collection 框架更一致。

14.8 如果存在冲突，向 Hash 表中插入的顺序是相关的。

14.9 平方探测的冲突概率小一些，因为它可以跳跃前进。不同于线性探测，平方探测在 Hash 表不满的情况下也有可能会导致无限循环。

14.10 HashMap 对象实现成带独立链和缺省负载系数为 75%的 Hash 表，因此，对于插入、删除和查找操作，它都提供 O(1)的访问时间。TreeMap 对象实现为平衡二叉搜索树，对于插入、删除和查找操作，它提供 O(lg n)的访问时间。因此，HashMap 速度快一些，但是 TreeMap 可以排序。

习 题 解 答

14.1 将 16 个表项插入到德英字典中：

```
import java.util.*;

public class Pr1401
{ public static void main(String[] args)
  { Map map = new HashMap(11);
    map.put("Ast","gate");
    map.put("Eis","ice");
    map.put("Hof","court, yard, farm");
    map.put("Hut","hat");
    map.put("Lob","praise");
    map.put("Mal","mark, signal");
    map.put("Mut","courage");
    map.put("Ohr","ear");
    map.put("Ost","east");
    map.put("Rad","wheel");
    map.put("Rat","advice, counsel");
    map.put("Tag","day");
    map.put("Tor","gate");
    map.put("Uhr","clock");
    map.put("Wal","whale");
    map.put("Zug","procession, train");
    System.out.println("map=" + map);
```

```
    System.out.println("map.keySet()=" + map.keySet());
    System.out.println("map.size()=" + map.size());
  }
}
```

14.2 能够过滤掉一些常用词汇的字句索引。

```
import java.io.*;
import java.util.*;

public class Pr1403
{ private static Map concordance;
  private static Set words;
  private static File file;
  private static FileReader reader;
  private static BufferedReader in;

  public static void main(String[] args)
  { loadSet("CommonWords.txt");
    dumpSet();
    loadMap("Shakespeare.txt");
    dumpMap();
  }
  private static void loadSet(String filename)
  { try
    { words = new HashSet();
      file = new File(filename);
      reader = new FileReader(file);
      in = new BufferedReader(reader);
      String line=null;
      StringTokenizer parser=null;
      while ((line=in.readLine()) != null)
      { parser = new StringTokenizer(line,",.;:()-!?' ");
        String word = parser.nextToken().toUpperCase();
        words.add(word);
      }
    }
    catch(Exception e) { System.out.println(e); }
  }

  private static void dumpSet()
  { for (Iterator it=words.iterator(); it.hasNext(); )
      System.out.println(it.next());
  }

  private static void loadMap(String filename)
```

标准分享网 www.bzfxw.com 免费下载

```
    { try
      { concordance = new TreeMap();
        file = new File(filename);
        reader = new FileReader(file);
        in = new BufferedReader(reader);
        String line=null;
        StringTokenizer parser=null;
        int lineNumber=0;
        while ((line=in.readLine()) != null)
        { ++lineNumber;
          parser = new StringTokenizer(line,",.;:()-!?' ");
          while (parser.hasMoreTokens())
          { String word = parser.nextToken().toUpperCase();
            if (!words.contains(word))
            { String listing = (String)concordance.get(word);
              if (listing==null) listing = "" + lineNumber;
              else listing += ", " + lineNumber;
              concordance.put(word,listing);
            }
          }
        }
      }
      catch(Exception e) { System.out.println(e); }
    }
    private static void dumpMap()
    { Set set = concordance.entrySet();
      for (Iterator it=set.iterator(); it.hasNext(); )
        System.out.println(it.next());
    }
}
```

14.3 带排序功能的德英字典：

```
import java.util.*;
public class Pr1202
{ private static Map map;

  public static void main(String[] args)
  { map = new TreeMap();
    load();
    dump();
  }

  private static void load()
  { map.put("Ast","gate");
    map.put("Eis","ice");
```

```
    map.put("Hof","court, yard, farm");
    map.put("Hut","hat");
    map.put("Lob","praise");
    map.put("Mal","mark, signal");
    map.put("Mut","courage");
    map.put("Ohr","ear");
    map.put("Ost","east");
    map.put("Rad","wheel");
    map.put("Rat","advice, counsel");
    map.put("Tag","day");
    map.put("Tor","gate");
    map.put("Uhr","clock");
    map.put("Wal","whale");
    map.put("Zug","procession, train");
  }

  private static void dump()
  { Set set = map.entrySet();
    for (Iterator it=set.iterator(); it.hasNext(); )
      System.out.println(it.next());
  }
}
```

14.4 程序如下：

```
import java.io.*;
import java.util.*;

public class Pr1404
{ private static Map concordance;
  private static File file;
  private static FileReader reader;
  private static BufferedReader in;

  public static void main(String[] args)
  { loadMap("Shakespeare.txt");
    dumpMap();
  }

  private static void loadMap(String filename)
  { try
    { concordance = new TreeMap();
      file = new File(filename);
      reader = new FileReader(file);
      in = new BufferedReader(reader);
```

标准分享网 www.bzfxw.com 免费下载

```
      String line=null;
      StringTokenizer parser=null;
      int lineNumber=0;
      while ((line=in.readLine()) != null)
      { ++lineNumber;
        parser = new StringTokenizer(line,",.;:()-!?' ");
        while (parser.hasMoreTokens())
        { String word = parser.nextToken().toUpperCase();
          String frequency = (String)concordance.get(word);
          if (frequency==null) frequency = "1";
          else
          { int n=Integer.parseInt(frequency);
            ++n;
            frequency = "" + n;
          }
          concordance.put(word,frequency);
        }
      }
    }
    catch(Exception e) { System.out.println(e); }
  }

  private static void dumpMap()
  { Set set = concordance.entrySet();
    for (Iterator it=set.iterator(); it.hasNext(); )
      System.out.println(it.next());
  }
}
```

第15章 集　合

15.1 数学集合

集合是不同元素的集合（collection）。集合的大小是该集合所包含元素的个数。元素个数为零的集合称为空集，用Φ表示。

在数学中，集合可用列举出其所有元素来表示，例如

$$A = \{44,77,22\}$$

或者指出元素呈现的模式（pattern），如：

$$B = \{2,4,6,8,10,12,..\}$$

或者指出元素满足的条件，如：

$$C = \{n \in \mathbf{Z} \mid n \text{ 是偶数}\} = \{n \in \mathbf{Z} : n \text{ 是偶数}\}$$

符号 **Z** 代表所有整数的集合：

$$\mathbf{Z} = \{0,1,-1,2,-2,3,-3,...\}$$

符号“|”“:”意思是“满足于”，那么上述集合 C 的定义就是“所有是偶数的整数 n 构成的集合”，其意思就是偶数集。

集合有 3 种基本运算：并、交、相对补（差）运算。集合 A 和集合 B 的并运算用 $A \cup B$ 表示。它的定义如下：

$$A \cup B = \{x: x \in A \text{ 或 } x \in B\}$$

集合 A 和集合 B 的交运算用 $A \cap B$ 表示。它的定义如下：

$$A \cap B = \{x: x \in A \text{ 且 } x \in B\}$$

集合 A 和集合 B 的差运算用 $A-B$ 表示。它的定义如下：

$$A-B = \{x: x \in A \text{ 但 } x \notin B\}$$

集合的基本关系算子是子集算子。A 是 B 的子集当且仅当 A 中的每个元素也是 B 中的元素。这可表示为 $A \subseteq B$。注意每个集合是它自身的子集，空集是任何集合的子集。

例 15.1　集合的典型运算

若 $A = \{1,2,3,4,5\}$，$B = \{4,5,6,7\}$ 则 $A \cup B = \{1,2,3,4,5,6,7\}$。$A \cap B = \{4,5\}$。$A-B = \{1,2,3\}$。

若 $C = \{2,3,4\}$，则 C 是 A 的子集，但不是 B 的子集。

标准分享网 www.bzfxw.com 免费下载

15.2 Java集合接口

在Java中集合数据结构用集合框架（Collections Framework）表示，这类同于表结构的表示（参见14章）。该框架包含如下类和接口，如图15.1所示。

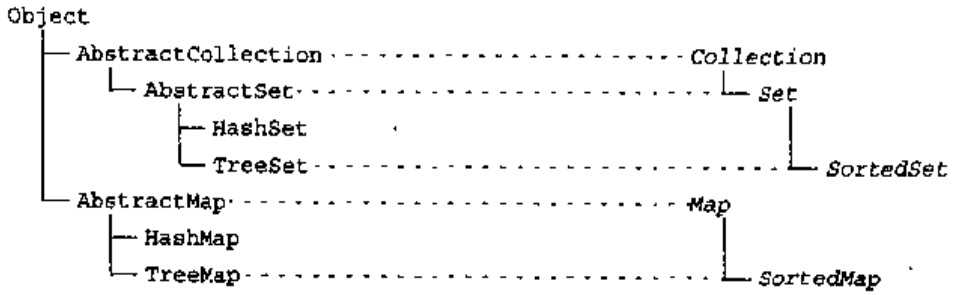


图15.1 Java集合框架的类（接口）层次图

java.util.Set接口是Collection接口的子接口，它继承如下方法：

```
public boolean  add(Object);
public boolean  addAll(Collection);
public void     clear();
public boolean  contains(Object);
public boolean  containsAll(Collection);
public boolean  equals(Object);
public int      hashode();
public boolean  isEmpty();
public Iterator iterator();
public boolean  remove(Object);
public boolean  removeAll(Collection);
public boolean  retainAll(Collection);
public int      size();
public Object[] toArray();
public Object[] toArray(Object[]);
```

Set接口中没有增加其他的方法。

15.3 Java AbstractSet 类

java.util.AbstractSet 类是 Set 接口的部分实现。正如集合框架中其他抽象类一样，设计该类的目的是减少创建一个完整类的开销。该抽象类实现了 Set 接口的大多数方法。

```
Public abstract class AbstractSet extends AbstractCollection
```

```
                                    implements Set
{  protected                        AbstractSet()  {}
   public      boolean              add(Object);
   public      boolean              addAll(Collection);
   public      void                 clear();
   public      boolean              contains(Object);
   public      boolean              containsAll(Collection);
   public      boolean              equals(Object);
   public      int                  hashCode()l
   public      boolean              isEmpty();
   public abstract Iterator iterator();
   public      boolean              remove(Object);
   public      boolean              removeAll(Collection);
   public      boolean              retainAll(Collection);
   public abstract int               size();
   public      Object()             toArray();
   public      Object()             toArray(Object[]);
}
```

15.4 Java HashSet 类

Java HashSet 类与 Java HashMap 类基本相同，二者的区别是 Java HashSet 类中的对象是单--的集合元素，而 Java HashMap 类中的对象是“关键字—数值对”两部分组成的映射（map）。与 HashMap 类似，HashSet 类利用对象自身的 hashCode()方法生成的值将对象存储在一个 hash 表中。因此可以在几乎相同的时间（算法复杂度为 $O(1)$）访问集合中的各元素（可通过 add()，contains()，remove()方法完成）。

该类在 java.util 包中的定义如下：

```
public class HashSet extends AbstractSet implements Set
{  public HashSet();
   public HashSet(Collection collection);
   public HashSet(int capacity,float threshold);
   public HashSet(int capacity);
   public Iterator iterator();
   public int size();
   public boolean isEmpty();
   public boolean contains(Object object);
   public boolean add(Object object);
   public boolean remove(Object object);
   public void clear();
   public Object clone();
}
```

标准分享网 www.bzfxw.com 免费下载

例 15.2 一个利用 HashSet 类的字符串操作

该程序创建了一个名为 countries 的 HashSet 对象，然后载入 7 个字符串，接着利用 iterator 方法遍历 hash 表，打印所有内容。

```
import java.io.*;
import java.util.*;

public class Ex1502
{ public static void main(String[] args)
  { Set countries = new HashSet();
    load(countries);
    dump(countries);
  }
    private static void load(Set set)
  { set.add("Cuba");
    set.add("Iran");
    set.add("Iraq");
    set.add("Laos");
    set.add("Oman");
    set.add("Peru");
    set.add("Togo");
  }

  private static void dump(Set set)
  { for (Iterator it=set.iterator(); it.hasNext(); )
      System.out.println(it.next());
    System.out.println("set.size() = " + set.size());
  }
}
```

上述程序的输出为：

```
Peru
Oman
Iraq
Iran
Cuba
Togo
Laos
set.size() = 7
```

注意在 hash 表中元素的顺序已被散列化（随机化）。元素在 hash 表中实际索引是通过一个专有的 hash 函数计算出，该函数是对字符串的 Object.hashCode() 的值模 101 得出，而 Object.hashCode() 又是通过字符的 Unicode 编码计算得来。

例15.3 修改一个HashSet类的字符串

本程序用于说明HashSet类的add()，contains()，remove()使用方法。该程序使用的数据同例15.2：

```
import java.util.*;

public class Ex1503
{ public static void main(String[] args)
  { Set countries = new HashSet();
    load(countries);
    report(countries,"Iraq");
    System.out.println("countries.remove(\"Iraq\") = "
                       + countries.remove("Iraq"));
    report(countries,"Iraq");
    report(countries,"Fiji");
    System.out.println("countries.remove(\"Fiji\") = "
                       + countries.remove("Fiji"));
    System.out.println("countries.add(\"Fiji\") = "
                       + countries.add("Fiji"));
    report(countries,"Fiji");
    System.out.println("countries.add(\"Fiji\") = "
                       + countries.add("Fiji"));
    report(countries,"Fiji");
  }

  private static void report(Set set, String string)
  { System.out.println("\t\t" + set.size() + " elements: " + set);
    System.out.println("\t\tset.contains(" + string + ") = "
                       + set.contains(string));
  }

  private static void load(Set set)
  { set.add("Cuba");
    set.add("Iran");
    set.add("Iraq");
    set.add("Laos");
    set.add("Oman");
    set.add("Peru");
    set.add("Togo");
  }
}
```

上述程序的输出为：

标准分享网 www.bzfxw.com 免费下载

```
            7 elements: [Peru, Oman, Iraq, Iran, Cuba, Togo, Laos]
            set.contains(Iraq) = true
countries.remove("Iraq") = true
            6 elements: [Peru, Oman, Iran, Cuba, Togo, Laos]
            set.contains(Iraq) = false
            6 elements: [Peru, Oman, Iran, Cuba, Togo, Laos]
            set.contains(Fiji) = false
countries.remove("Fiji") = false
countries.add("Fiji") = true
            7 elements: [Peru, Fiji, Oman, Iran, Cuba, Togo, Laos]
            set.contains(Fiji) = true
countries.add("Fiji") = false
            7 elements: [Peru, Fiji, Oman, Iran, Cuba, Togo, Laos]
            set.contains(Fiji) = true
```

remove()方法返回成功或失败，因此 countries.remove("Fiji")返回假，因为在当时集合中不包括"Fiji"。同样 add()方法也返回成功或失败。这样访问集合元素（通过 add()、remove()、contains()方法）的时间都是对数量级（算法复杂性为 $O(\lg n)$）。

15.5 Java TreeSet 类

Java TreeSet 类与 Java TreeMap 类基本相同，二者的区别是 Java TreeSet 类中的对象是单一的集合元素，而 Java TreeMap 类中的对象是关键字－数值对两部分组成的映射。与 TreeMap 类似，TreeSet 类将对象存储在一个平衡二叉搜索树中（一个红-黑树）。访问集合元素（通过 add()，remove()，contains()方法）的时间是对数量级（算法复杂性为 $O(\lg n)$）。

该类在 java.util 包中的定义如下：

```
public class TreeSet extends AbstractSet implements Set
{  public TreeSet();
   public TreeSet(Collection collection);
   public TreeSet(SortedSet set);
   public TreeSet(Comparator comparator);
   public Iterator iterator();
   public int size();
   public boolean isEmpty();
   public boolean contains(Object object);
   public boolean add(Object object);
   public boolean remove(Object object);
   public void clear();
   public Object clone();
```

```
    public comparator comparator();
    public Object first();
    public Object last();
    public SortedSet subSet(Object start,Object stop);
    public SortedSet headSet(Object stop);
    public SortedSet tailSet(Object start);
}
```

TreeSet 类包含了与 HashSet 类中相同的方法，额外的方法是通过 SortedSet 接口实现的，包括上述列举的后 5 种方法，并相应返回第一个元素，最后一个元素，以及 3 种对应子集。

HashSet 类使用 equals()方法搜寻待查的元素，这已经足以满足要求，因为查找的顺序在一个 hash 表中是无关紧要的。然而这在搜索树中却是非常重要的。因此 TreeSet 类利用元素的 compareTo()方法或 Comparator 对象来代替 equals()方法。如果激活上面所列的第 4 个构造函数，则可使用 Comparator 对象。

例 15.4 利用 TreeSet 类的字符串

该程序用于说明 TreeSet 类中 first()，last()，headSet()，subSet()，tailSet()的使用方法。

```
import java.util.*;
public class Ex1504
{ public static void main(String[] args)
  { SortedSet countries = new TreeSet();
    load(countries);
    System.out.println(countries);
    System.out.println("countries.first() = " + countries.first());
    System.out.println("countries.last()  = " + countries.last());
    String s1="Fiji";
    String s2="Laos";
    System.out.println("countries.headSet(" + s1 + ") = "
                       + countries.headSet(s1));
    System.out.println("countries.subSet(" + s1 + "," + s2 + ") = "
                       + countries.subSet(s1,s2));
    System.out.println("countries.tailSet(" + s2 + ") = "
                       + countries.tailSet(s2));
  }

  private static void load(Set set)
  { set.add("Laos");
    set.add("Cuba");
    set.add("Iraq");
    set.add("Togo");
```

标准分享网 www.bzfxw.com 免费下载

```
        set.add("Iran");
        set.add("Oman");
        set.add("Chad");
        set.add("Peru");
        set.add("Guam");
        set.add("Fiji");
    }
}
```

上述程序的输出为：

```
[Chad, Cuba, Fiji, Guam, Iran, Iraq, Laos, Oman, Peru, Togo]
countries.first() = Chad
countries.last()  = Togo
countries.headSet(Fiji) = [Chad, Cuba]
countries.subSet(Fiji,Laos) = [Fiji, Guam, Iran, Iraq]
countries.tailSet(Laos) = [Laos, Oman, Peru, Togo]
```

注意：结果集合的元素范围的规则是，它的起始元素是待查子集的第一个元素，结尾元素是待查子集最后元素的前一个。例如 countries.subSet("Fiji","Laos")调用返回的集合中第一个元素是"Fiji",最后一个元素是"Iraq"，"Iraq"是"Laos"的前一个元素。

复 习 题

15.1 在一个已包含某元素的集合中，调用 add()加入该元素，结果如何？

15.2 从一个不包含某元素的集合中，调用 remove()删除该元素，结果如何？

15.3 请比较 HashSet 与 TreeSet 的优缺点。

习 题

15.1 用 TreeSet 对象实现下述集合的典型操作：

```
public static Set union(Set a, Set b)
//返回一个新的集合对象，该对象包含且只包含集合 a 中的元素和集合 b 中的元素
```

15.2 用 TreeSet 对象实现下述集合的典型操作：

```
public static Set intersection(Set a, Set b)
//返回一个新的集合对象，该对象包含且只包含集合 a 和集合 b 中的公共元素
```

15.3 用 TreeSet 对象实现下述集合的典型操作：

```
public static Set complement(Set a, Set b)
//返回一个新的集合对象，该对象包含且只包含出现在集合 a 中
//但不出现在集合 b 中的元素
```

15.4 用 TreeSet 对象实现下述集合的典型操作：

```
public static Set isSubset(Set a, Set b)
//如果集合 a 中的每个元素都包含在集合 b 中，返回 true，否则返回 false
```

复习题答案

15.1 在一个已包含某元素的集合中，调用 add()加入该元素，结果返回失败 false。（参见例 15.3）

15.2 从一个不包含某元素的集合中，调用 remove()删除该元素，结果返回失败 false。（参见例 15.3）

15.3 HashSet 对象是通过 hash 表实现，而 hash 表又是由多个单独的链表组成，它的缺省命中率是 75%，所以访问集合元素（插入、删除、查找）的时间接近 $O(1)$。TreeSet 对象通过平衡二叉搜索树实现，访问集合元素（插入、删除、查找）的时间为 $O(\lg n)$。所以 HashSet 访问对象快，而 TreeSet 对象是排了序的。

习 题 解 答

15.1 TreeSet 对象的并操作：

```
public static Set union(Set a, Set b)
{ Set c = new TreeSet(a);
  for (Iterator it=b.iterator(); it.hasNext(); )
    c.add(it.next());
  return c;
}
```

15.2 TreeSet 对象的交操作：

```
public static Set intersection(Set a, Set b)
{ Set c = new TreeSet(a);
  for (Iterator it=a.iterator(); it.hasNext(); )
  { String x=(String)it.next();
```

标准分享网 www.bzfxw.com 免费下载

第16章 图

16.1 简单图

简单图是一对序偶 $G=(V, E)$，其中 V 和 E 是有限集合，集合 E 的每一个元素是集合 V 的二元素子集（也就是 V 中不同元素的二元对）。V 中的元素称为点，E 中的元素称为边（或弧）。若 $e\in E$ 则 $e=\{a,b\}$ $a, b\in V$。在这种情况下，我们简单标识 e 为 $e=ab=ba$。我们说边 e 连接了两点 a 和 b，也就是说 e 与 a，b 相关联，或者说 a 和 b 关联于 e，或者说 a 和 b 是边 e 的两端点，或者说 a 和 b 相邻。

图的大小是指顶点集合中元素的个数。

例 16.1 简单图

图16.1给出了一个大小为4的简单图(V, E)。它的顶点集合是$V=\{a, b, c, d\}$，它的边集合是 $E=\{ab, ac, ad, bd, cd\}$。该图有4个顶点和5条边。

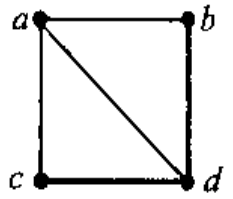
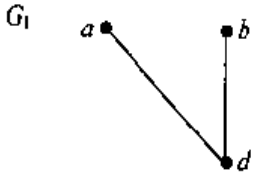


图 16.1 大小为4的简单图

注意根据定义，一条边是两顶点元素的集合。这可防止边的回路，因为一个回路仅涉及一个顶点。所以简单图的定义排除了回路的可能性。

还需注意的是由于 E 是一个集合，所以 E 中的边不可能出现两次。（集合不允许出现重复元素）所以简单图的定义排除了多条边重复的可能性。

通常，图允许回路和多条边重复；而简单图不允许。



16.2 图的术语

若 $G=(V, E)$是图，$G'=(V', E')$，且满足 $V'\subseteq V$ 和 $E'\subseteq E$，则 G' 称为 G 的子图。若 $V'=V$，则称 G'为 G 的生成子图。

每一个图都是自身的生成子图。

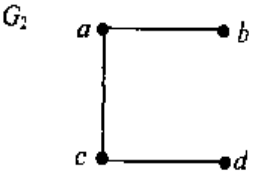


图 16.2 子图

例 16.2 子图

图 16.2 给出子图的例子。

图 $G_1=(V_1, E_1)$的顶点集合是 $V_1=\{a, b, d\}$，边集合是 $E_1=\{ad, bd\}$，它是例 16.1 的非生

成子图。该子图的大小为 3。

图 $G_2=(V_2, E_2)$的顶点集合是 $V_2=\{a, b, c, d\}$，边集合是 $E_2=\{ad, ac, cd\}$，它是例 16.1 的生成子图。该子图的大小为 4。

顶点的度是指在该顶点相交的边的个数。例如，在例 16.1 的图中点 a 的度是 3，点 b 的度是 2。在例 16.2 的子图中，点 a 和点 b 的度是 1。

孤立点是指度为 0 的点。

定理 16.1　一个边数为 m 的图中，所有顶点的度之和等于 $2m$。

证明：由于每一条边都使得该边的两顶点的度各增加 1，所以 m 条边就使得图中所有顶点的度之和等于 $2m$。

一个完全图是指一种简单图，它满足图中任意两点之间都有一条边。对于顶点数为 n 的图，只有一个与之对应的完全图。因此我们说一个完全图均是指基于某个顶点集合的完全图。

例 16.3　4 个点的完全图

图 16.3 是一个基于集合 $V=\{a,b,c,d\}$的完全图，它的边集合 E 为 $E=\{ab, ac, ad, bc, bd, cd\}$。

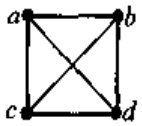


图 16.3　完全图

定理 16.2　n 个点的完全图的边的个数等于 $n(n-1)/2$。

证明：有 n 个点，而每个点都与其他 $n-1$ 个点相邻。所以就有 $n(n-1)$个有序点对。故有 $n(n-1)/2$ 个无序点对；因为每一个无序点对可以有两种有序方式。例如一个无序对$\{a, b\}$可有两种有序方式：(a, b)或(b, a)。

例如，在例 16.3 中 4 点的完全图的边的个数等于 $n(n-1)/2=4(4-1)/2=6$。

推论 16.1　n 个点的任何图的边的个数 $m\leq n(n-1)/2$。

16.3　路径与回路

在图中从点 a 到点 b 的路径是指边的序列$(a_0a_1,a_1a_2,\ldots,a_{k-1}a_k)$，满足 $a_0=a$ 和 $a_k=b$；也就是说，存在一个边序列$(e_1,e_2,\ldots,e_k)$，边 e_i 连接某个顶点和点 a_i，而下一条边 e_{i+1} 连接点 a_i 和某个其他点，因此最终形成从点 a 到点 b 的一条链。一条路径的长度是指形成该路径的边的个数 k。

尽管一条路径是边的序列，但很自然得出这是由边连接的相邻顶点组成的序列。所以只要在图中(a_{i-1}, a_i)是一条边，我们将路径$(a_0a_1,a_1a_2,\ldots,a_{k-1}a_k)$简单地标识为 $a_0a_1a_2\ldots a_{k-1}a_k$。

若 $p = a_0a_1a_2,\ldots a_{k-1}a_k$ 是图中的一条路径，我们说 p 是从 a_0 到 a_k（或从 a_k 到 a_0）的一条路径；我们也可以说 p 连接点 a_0 和 a_k 或 a_0 和 a_k 被 p 连接。a_0 和 a_k 称为该条路径的两端点。

一条路径是指所包含顶点都不同的路径。

标准分享网 www.bzfxw.com 免费下载

例 16.4 图的路径

如图 16.4 所示，*abcfde* 是一条长为 5 的路径。正规地说，该路径是(*ab*,*bc*,*cf*,*fd*,*de*)。路径 *abefdbc* 的长度为 6，但它不是一条路径，因为点 *b* 出现了两次。同样，长度为 4 的路径 *abefa* 也不是路径。序列 *abf* 不是一条路径，因为 *bf* 不是一条边。序列 *abb* 也不是一条路径，因为 *bb* 不是一条边。*aba* 是长度为 2 的路径，*ab* 是长度为 1 的路径。

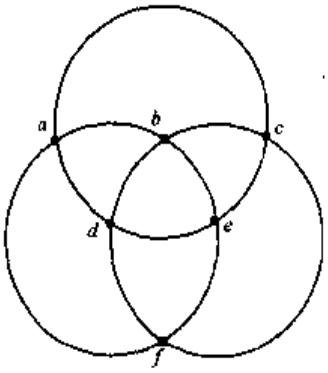


图 16.4 图的路径

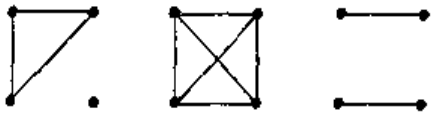
我们说一个图是连通的是指图中任何两点之间存在连接的路径，否则称该图为非连通。

前面给出的所有图都是连通的。

例 16.5 一个非连通图

图 16.5 是一个大小为 12 的非连通图。

一个图的连通部分是指最大连通的子图；也就是说一个连通的子图有这样的特性：任何包含它，比它再大的子图均是非连通的。


16.5 连通图

例 16.5 的图中有 5 个连通部分：大小分别是 3，1，4，2 和 2。

定理 16.3 每一个图是连通部分的惟一组合。

若一条路径的两端点是同一个点，则该路径是闭环的。一条回路是长度至少为 3 的闭环路径，且中间的各顶点不同。

例 16.6 图的回路

在例 16.4 的图中，路径 *abefa* 是一条回路。路径 *abedbcfa* 不是一条回路，因为它不是一条路径：中间出现重复顶点 *b*。路径 *abef* 也不是一条回路，因为它不是闭环的。

路径 *aba* 不是回路，因为它的长度只有 2。

一个图若不包含任何回路，则称为非循环的。

在上面所给的图中，只有例 16.2 的图是非循环的。

一个非循环图也称为一个自由森林，一个连通的非循环图称为自由树。注意一个树与一个自由树是相同的，其中都有一个点被指定为根。所以，在图论中，一个树也称为一个有根的树，它的定义是有一个特殊节点的连通非循环图。

一个图的生成树是一个连通非循环生成子图。

例 16.7 生成树

图 16.6 中，右边部分是左边部分的生成树。

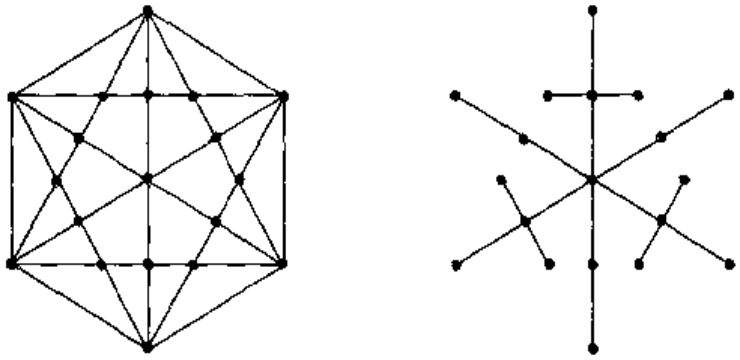


图 16.6　生成树

16.4　同构图

图 $G=(V, E)$和图 $G'=(V', E')$是同构的是指存在函数 f 将 V 中的每个点 x 对应到 V'中的某个点 $y=f(x)$上，并满足下面 3 个条件：

1. f 是一对一的：V 中的每个点 x 对应 V'中的不同点 $y=f(x)$。

2. f 是可逆的，V'中的每个点 y 对应 V 中的某个点 x。

3. f 保持连接性：若$\{x_1,x_2\}$是 E 中的某条边，则$\{f(x_1),f(x_2)\}$是 E'中的某条边。

若从一个图到另一个图存在同构函数，则这两个图是同构的。“同构”的含义是同样的样式。对于图来说，是指这两图具有相同的拓扑结构。若一个图不用破坏任何边的连接，只是通过线的扭拉变换，则这两图是同构的。

例 16.8　同构图

图 16.7 所示的两个图是同构的。

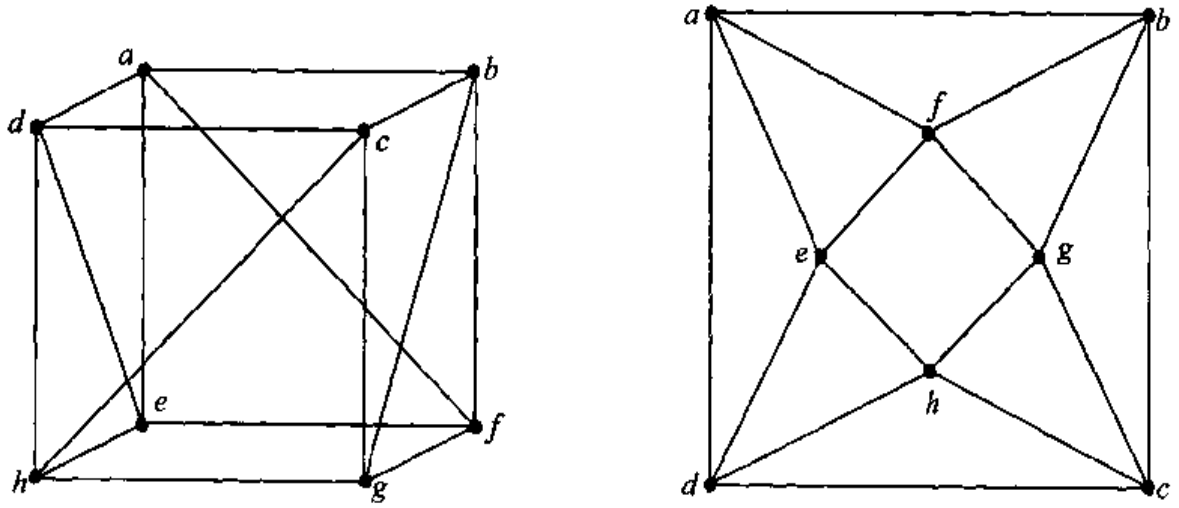


图 16.7　同构图

标准分享网 www.bzfxw.com 免费下载

通过对顶点作标签来发现两图的同构映射关系。

可以这样证明：如果在一个图中，点 x_1 与点 x_2 是邻接关系，那么对应点在另一图中也是邻接关系。例如点 a 在两个图中与点 b，d，e，f（但与 c，g，h 不邻接）邻接。根据定义证明两图是同构的，必须要找到两图之间的同构函数。这等同于用相同的标签集合对两图作标签，使得两图的标签之间具有相同的邻接关系。由于有 $n!$种不同情况，很快发现这样的同构函数几乎是不可能的。例如对于例 16.8 中的图用 8 个标签对图中的 8 个点做标识，有 8!=40 320 种不同方法。下面算法更有效：

1. 对一个图中的点任意做标签（假设用正整数做标签）。
2. 在第 2 个图中寻找与第 1 个图中点 1 有相同度数的点，并用 1 标识。
3. 对第二个图中点 1 邻接的点做标签，标号与第 1 个图中点 1 邻接点的标签相同。
4. 对每个新标识的点重复第 3 步。

如果在某点，第三步变为不可能，则回溯并重新进行不同的标识。如果到某一步，无法回溯，则可试图证明两图是不同构的。

根据定义证明两图是不同构的，需要列举每一种不同的作标识的方法，$n!$种都不能保持点的邻接关系。这样的证明方法是不切实际的。下面给出的定理使得证明两图不同构更加容易。

定理 16.4　图的同构性证明。

下面的条件是证明两图是同构的必要条件：

1. 它们必须有相同的顶点数目。
2. 它们有相同的边数。
3. 它们有相同的连通部分。
4. 同度数顶点的数目相同。
5. 同长度的回路的数目相同。

例 16.9　证明两图非同构

图 16.8 中有 3 个图，将它们与例 16.8 的两个同构图相比较：

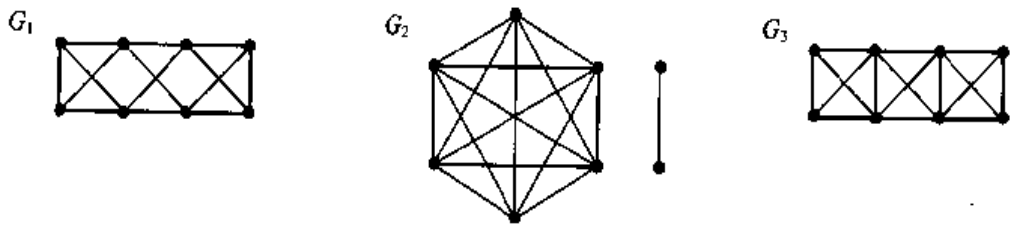


图 16.8　图的同构性证明

每一个图有 8 个顶点，所以每个图可能与例 16.8 的两个图同构。

图 G_1 不可能与那两个图同构，因为它有 14 条边。例 16.8 的两个图各有 16 条边。定理 16.5 的条件 2 指出同构图必须要有相同的边数。

图 G_2 有 16 条边。但它与那两个图不同构，因为它有 2 个连通部分。而例 16.8 的两个图都只有一个连通部分，定理 16.5 的条件 3 指出同构图有相同的连通部分。

图 G_3 有 16 条边，有 1 个连通部分。但它与那两个图不同构，因为它有度数为 3 和度数为 5 的点，而例 16.8 的两个图的点的度数都为 4，定理 16.5 的条件 4 指出同构图同度数顶点的数目相同。

注意例 16.9 是将每个图与例 16.8 中的一个图做比较，而不是对两个图同时做比较。

定理 16.5　图的同构性是相等关系。

图的同构关系满足相等关系的 3 个特性：

1. 每个图与自身是同构的。
2. 若 G_1 与 G_2 同构，则 G_2 与 G_1 同构。
3. 若 G_1 与 G_2 同构，G_2 与 G_3 同构，则 G_1 与 G_3 同构。

推论 16.2

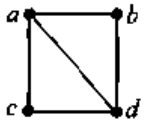
若 G_1 与 G_2 同构，某图 G 与 G_1 不同构，则图 G 与 G_2 也不同构。

16.5　图的邻接矩阵

图的邻接矩阵是一个二维布尔数组

```
boolean [] [] a;
```

对于点集 $V=\{v_0,v_1,\ldots,v_{n-1}\}$，当且仅当点 v_i 与 v_j 邻接，a[i][j]的值为真。



	a	b	c	d
a	F	T	T	T
b	T	F	F	T
c	T	F	F	T
d	T	T	T	F

图 16.9　例 16.1 的图的邻接矩阵

例 16.10　一个邻接矩阵

图 16.9 是例 16.1 的图的邻接矩阵。

注意，邻接矩阵有下列性质：

1. 矩阵是对称的。也就是说对所有的 i 和 j，a[i][j]=a[j][i]。
2. 矩阵中为真的项的个数是边的个数的两倍。
3. 对同一个图，集合 V 的元素的顺序不同，邻接矩阵不同。

	a	b	c	d
a	0	1	1	1
b	1	0	0	1
c	1	0	0	1
d	1	1	1	0

图 16.10　例 16.10 的邻接矩阵

邻接矩阵中的项，我们通常用 0 代替假，1 代替真。用这种形式，则例 16.10 的邻接矩阵如图 16.10 所示。

标准分享网 www.bzfxw.com 免费下载

16.6 图的关联矩阵

图(V, E)的关联矩阵是一个二维数组

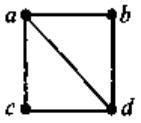
```
int[][]a;
```

对于点集合$V=\{v_0,v_1,\ldots,v_{n-1}\}$和边集合$E=\{e_0,e_1,\ldots,e_{m-1}\}$，当点$v_i$与边$e_j$相关联时，a[i][j]为 1，否则为 0。

例 16.11 一个关联矩阵

图 16.11 是例 16.1 图的关联矩阵。第一行指出点 a 与边 1，2，3 关联；第二行指出点 b 与边 1，4 关联等等。

注意：无论一个图有多少点和边，任何关联矩阵的每一列都有且只有两项为 1。为什么？（参见习题 16.6）



a	1	1	1	0	0
b	1	0	0	1	0
c	0	1	0	0	1
d	0	0	1	1	1

图 16.11 例 16.1 图的关联矩阵

16.7 图的邻接表

图（V, E）的邻接表（或邻接结构）是包含若干元素的列表，每个元素是图中的每个顶点的相邻点的列表。对每个顶点的相邻点的列表又称为边表。

例 16.12 邻接表

图 16.12 是例 16.1 图的邻接表。

点 a 的边表有 3 个元素，每个都是与点 a 相关联的边；点 b 的边表有两个元素，每个都是与点 b 相关联的边，依次类推。

注意，每一个边表中的元素对应该图的关联矩阵的某一项为真。例如点 c 的边表有两个元素对应在例 16.11 中的关联矩阵的第三行有 2 项为真。所有的边表组成的集合与关联矩阵有相同的数组形式。

例 16.12 的图表相对复杂些。尽管在边表中存贮着点，我们一般直接列出相邻的点来代替非直接的表示，如图 16.13 所示。

注意：边表是无序的，也就是说它们的顺序是无关

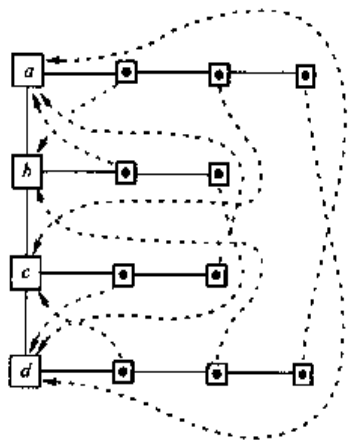


图 16.12 例 16.1 图的邻接表

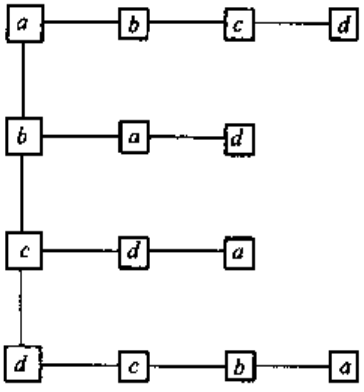


图 16.13 邻接表的另一种表示

紧要的。

16.8　有向图

有向图是一个序偶 $G=(V, E)$，V 是一个有限集合，E 是 V 中两元素组成的有序对的集合。与非有序图类似，V 中的元素称为点，E 中的元素称为边。若 $e\in E$，则 $e=(a,b)$满足 $a,b\in V$。简单起见，用 $e=ab$ 来标识 e。我们说边 e 起始于点 a，终止于点 b。一个点的出度是指起始于该点的边数，　个点的入度是指终止于该点的边数。

注意：与图的定义不同，有向图的定义很自然允许一条边起始于某点，终止于相同点，这种情形，称为回路。一个简单有向图是无回路的有向图。

例 16.13　一个有向图

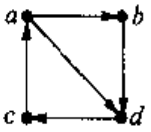


图 16.14　有向图

图 16.14 是一个有向图，点集 $V=\{a,b,c,d\}$，边集合 $E=\{ab,ad,bd,ca,dc\}$。

点 a 的出度为 2，入度为 1。点 b 和点 c 的入度、出度都为 1。点 d 的出度是 1，入度是 2。

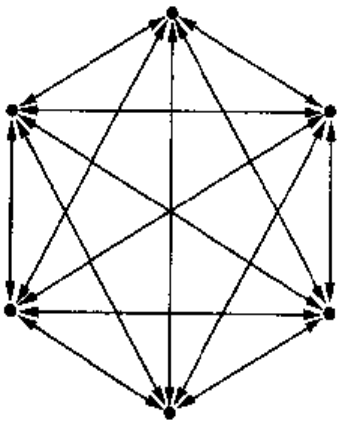
定理 16.6　若 G 是有 m 条边的有向图，则出度的总数等于 m，入度的总数等于 m。

证明：每一条边都为出度增加 1，为入度增加 1，所以出度、入度的总和各等于 m。

完全有向图是指一个有向图的每个点都有到其他点的一条有向边。

例 16.14　6 个点的完全有向图

图 16.15 是 6 个点的有向图。它有 15 条双向边。所以总的边数是 30，$n(n-1)=6(6-1)=30$。


图 16.15　完全有向图

定理 16.7　n 个点的完全有向图的边数是 $n(n-1)$。

证明：根据定理 16.2，n 个点的完全非有向图有 $n(n-1)/2$ 条非有向边。也就是所说 $n(n-1)/2$ 对有向边。所以 n 个点完全有向图的边数是它的两倍，$n(n-1)$.。

推论 16.3　任何 n 个点的有向图的边数 $m\leq n(n-1)$。

每一个有向图有一个嵌入图，将每一条有向边转换为无向边，并去掉重复边和环路就得到一个有向图的嵌入图。从数学上说，这就是将 E 中的有序对(x,y)转换为集合$\{x,y\}$，并去除所有大小为 1 的集合（也就是去除单元素）。

例 16.15　一个有向图的嵌入图

例 16.13 有向图的嵌入图如图 16.16 所示。

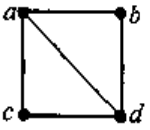


图 16.16　有向图的嵌入图

标准分享网 www.bzfxw.com 免费下载

有向图(*V*, *E*)的邻接矩阵是一个二维的布尔数组

```
boolean[][]a;
```

数组的值是这样得来的：对点集 $V=\{v_0,v_1,\ldots v_{n-1}\}$排序，当且仅当存在以点 v_i为起始点，点 v_j为终点的边时，a[i][j]为真。

例 16.16 一个有向图的邻接矩阵

图 16.17 是例 16.13 图的邻接矩阵。

注意：一个有向图的邻接矩阵中为真的项的个数等于图中边的个数。与无向图一样，点集合 *V* 中不同的元素顺序，会导致相同图有不同的邻接矩阵。

	a	*b*	*c*	*d*
a	F	T	F	T
b	F	F	F	T
c	T	F	F	F
d	F	F	T	F

图 16.17 有向图的邻接矩阵

一个有向图(*V*, *E*)的关联矩阵是一个二维的整型数组

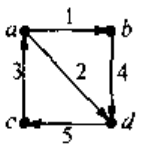
```
int[][]a;
```

数组的值是这样分配的：

对点集合 $V=\{v_0,v_1,\ldots v_{n-1}\}$排序，边集合 $E=\{e_0,e_1,\ldots e_{m-1}\}$排序，然后，当存在以点 v_i为起始点，点 v_j为终点的边时，a[i][j]等于 1，a[j][i]等于–1，其他情况为 0。

例 16.17 一个有向图的关联矩阵

图 16.18 是例 16.13 有向图的关联矩阵。



a	1	1	0	-1	0
b	-1	0	1	0	0
c	0	0	0	1	-1
d	0	-1	-1	0	1

图 16.18 有向图的关联矩阵

矩阵的第一行表明有两条边以点 *a* 作为起始点，有一条边以点 *a* 作为终点。

最后一个 1 在点 *d* 那行，且在最后一列。最后一列的非 0 项是对应点 *c* 行的–1，它表示有一条边以点 *d* 为起始点，点 *c* 为终点。

一个有向图(*V*,*E*)的邻接表是一个列表，表中包含的每一项是针对图中的每个点；每个点列表又包括以该点为起始点的边的列表。这与一个图的邻接表一样，只是这些边是单向的，与图中的边不分方向不同。

例 16.18 一个有向图的邻接表

图 16.19 是例 16.13 有向图的邻接表。点 *a* 的边表有两个元素，这表明有两条以点 *a* 为起始点的

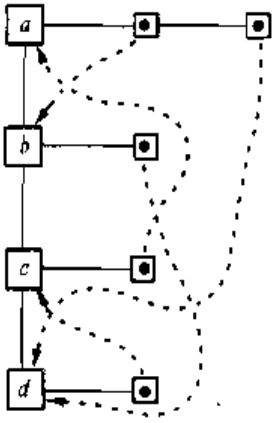


图 16.19 有向图的邻接表

边：*ab* 和 *ad*。

16.9　有向图的路径

在一个有向图中从点 *a* 到点 *b* 的路径是指一系列边$(a_0a_1,a_1a_2,\ldots,a_{k-1}a_k)$，满足 $a_0=a,a_k=b$。

与非有向图中的非有向路径一样，有向路径也可简写为点串：$p = a_0a_1a_2\ldots a_{k-1}a_k$，也可以说路径起始于点 *a*，终止于点 *b*。

如果一条路径的起始点和终止点是相同的，则称该路径是闭环的。一条路径是中间点各不同的路径。一条回路是中间点各不同的闭环路径。

例 16.19　有向路径

在例 16.13 中的有向图中，*adcabdc* 是长为 6 的非闭环路径。路径 *abdcacda* 是闭环的，但它不是回路，因为点 *d* 和点 *c* 在中间点中重复。路径 *dcab* 是一条非闭环路径。路径 *cabdc* 是长度为 4 的回路，*dcad* 是长度为 3 的回路。

注意不同的回路可能遍历相同的点。例如，例 16.13 的有向图中 *adca* 和 *cadc* 是不同的回路，但遍历的点相同。

一个有向图的任意两点间存在一条路径，则称该图是强连通的。一个有向图的嵌入图是连通的，则称该图是弱连通的。一个有向图如果上述两者都不符合，则称为非连通的。

例 16.20　强连通与弱连通有向图

如图 16.20 所示，有向图 G_1 是强连通的（它也是弱连通）。有向图 G_2 是弱连通而不是强连通，因为不存在以点 *x* 为终点的路径。有向图 G_3 是非连通的。

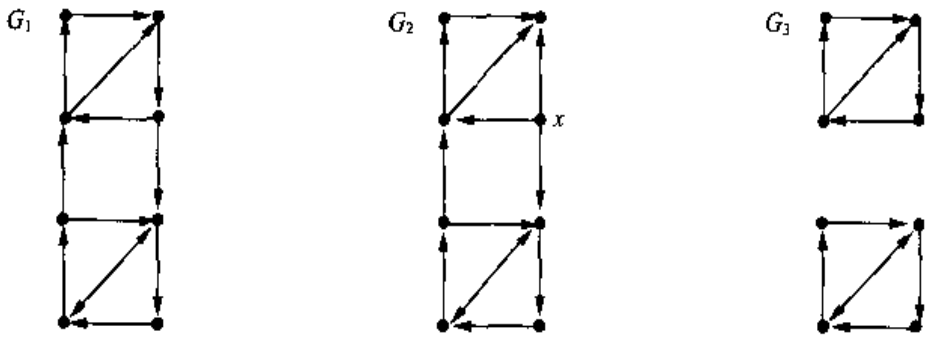


图 16.20　强连通与弱连通有向图

16.10　加权有向图和加权图

一个加权有向图是一对序偶(V,w)，这里 *V* 是一个有限点集，*w* 是一个函数，它给每个

标准分享网 www.bzfxw.com 免费下载

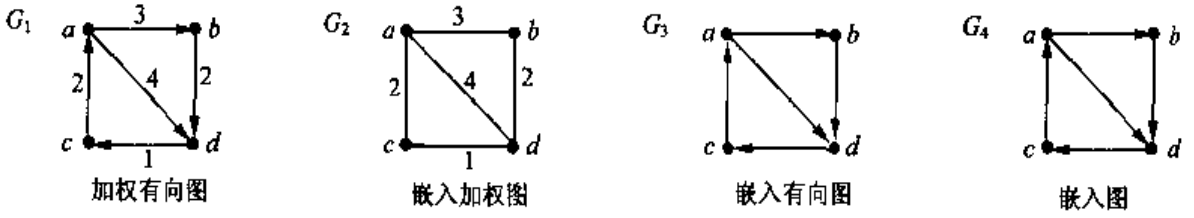
有序点对(x,y)分配一个正整数或无穷大∞。函数 w 称为加权函数，它的值 $w(x,y)$可以解释为从 x 到 y 的开销（时间或距离）。值 $w(x,y)=\infty$表示从 x 到 y 没有边。

一个加权有向图的加权函数 w 是对称的：也就是说对于所有 $x,y\in V$，满足 $w(y,x)=w(x,y)$。对于每个有向图有一个嵌入图，每个加权有向图有一个嵌入加权图(V,w)和一个嵌入非加权图。一个嵌入加权图的加权函数可以这样定义 $w'(x,y)=\min\{w(x,y),w(y,x)\}$，$w$ 是加权有向图的加权函数。嵌入有向图的边集合可以这样定义：$E=\{(x,y):w(x,y)<\infty\}$。

上面描述的有向图和图的特性适用于加权有向图和加权图。很显然由于加权函数，加权有向图和加权图也增加了一些特性。例如，加权路径长是一条路径上所有边的权值之和。从 x 到 y 的最短路径是所有从 x 到 y 的路径中加权路径长最短的那条路径。

例 16.21　一个加权有向图和它的嵌入结构

图 16.21 是一个加权有向图和它的嵌入加权图，嵌入有向图以及嵌入图。权值标识在各条边上。



16.21　加权有向图与其嵌入结构

在图 G_1 中，路径 $cabd$ 的加权路径长是 $|cabd|=2+3+2=7$，从 c 到 d 的最短路径是 6（沿着路径 cad）。然而在图 G_2 中，最短路径是 1（沿着路径 cd）。

注意图 G_3 与例 16.13 图一样，图 G_4 与例 16.1 图一样。

图 16.22 是图 G_1 的邻接矩阵、关联矩阵和邻接表。

	a	b	c	d
a	∞	3	∞	4
b	∞	∞	∞	2
c	2	∞	∞	∞
d	∞	∞	1	∞

a	3	4	0	-2	0
b	-3	0	2	0	0
c	0	0	0	2	-1
d	0	-4	-2	0	1

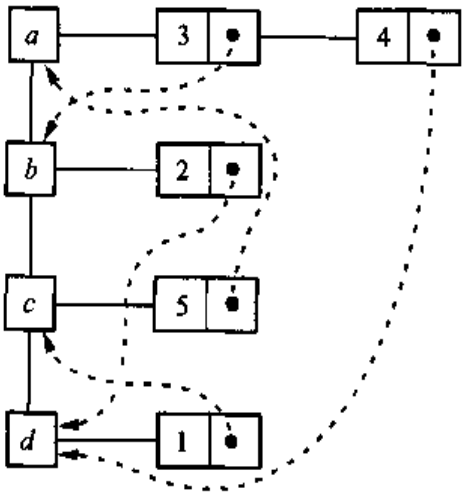


图 16.22　图 G_1 的邻接矩阵、关联矩阵和邻接表

16.11　欧拉路径、欧拉回路、哈密顿路径、哈密顿回路

在一个图中，一条欧拉路径是指仅包含每条边一次的路径。一条欧拉回路是指仅包含每条边一次的闭环路径。存在一条欧拉回路的图称为欧拉图。

注意：欧拉路径与欧拉回路不需要包含不同的点，因此不是严格意义上的路径。

例 16.22　欧拉路径和欧拉回路

在图 16.23 中，闭环路径 *acedabefdbcfa* 是一条欧拉回路，所以该图是一个欧拉图。

注意：图中每个顶点的度是 4，它的 12 条边可以划分为 3 个回路。根据下面定理的描述，这两个特性保证该图是一个欧拉图。

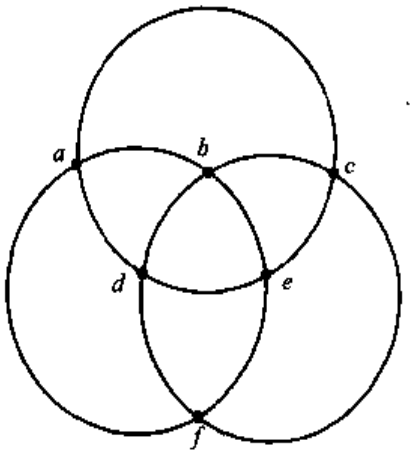


图 16.23　欧拉图

定理 16.8　欧拉图

若 G 是一个连通图，则下述条件等价：

1. G 是欧拉图。
2. 每个顶点有偶数的度。
3. G 中所有边的集合可以划分为几个回路。

在一个图中，一条哈密顿路径是指仅包含每个点一次的路径。一条哈密顿回路是指仅包含每个点一次的闭环路径。存在一条哈密顿回路的图称为哈密顿图。

不幸的是对于哈密顿图没有像定理 16.8 那样简单的判定规则。事实上，哈密顿图的判定规则仍是计算机科学中尚未解决的一个问题。

例 16.23　哈密顿图

如图 16.24 所示，左边的图是一个哈密顿图，右边的图不是，它有一条哈密顿路径，但没有哈密顿回路。



图 16.24　哈密顿图

16.12　DIJKSTRA 算法

Dijkstra 算法是用于在一个有向图中求从一个点 v_0 到其他点的最短路径。当算法完成时，

标准分享网 www.bzfxw.com 免费下载

从点 v_0 到 v 的最短路径长存储在点 v 中，从点 v_0 到 v 的最短路径记录在该路径上点 v 的前面各点中（参见例 16.24）。算法使用一条优先级队列，先对每个点初始化，然后对每个点重复进行出对操作。

算法 16.1 Dijkstra 的最短路径算法

（前提：$G=(V,w)$是一个加权有向图，并有初始点 v_0）

（结果：在图中每个点 v 存储从点 v_0 到点 v 的最短路径，该最短路径的上一个点可通过向后引用找到）

1. 点 v_0 的路径初始化为 0，到其他各点的路径初始化为∞。
2. 将所有的点在优先级队列 Q 中排队，路径长度越小，优先级越高。
3. 重复步骤 4～10，直到队列 Q 为空。
4. (不变量：每个点的路径长度、向后引用不在队列 Q 中是正确的)
5. 将优先级最高的点弹出队列，置于 x 中。
6. 对于优先级队列中点 x 的邻接点 y 操作步骤 7～10。
7. 让 s 等于 x 的路径值加上从 x 到 y 的边的权值。
8. 若 s 小于 y 的路径值，作步骤 9～10，否则返回步骤 3。
9. 将 s 作为 y 的路径值。
10. 将 x 作为 y 的向后引用点。

例 16.24 跟踪 Dijkstra 算法

图 16.25～图 16.29 是算法 16.1 用于 8 个点的图的运行轨迹。每一次跌代过程中仍留在优先级队列中的点用阴影标识，点 x 单独标识。每个点的距离域附在每个点的旁边，向后引用点用箭头表示。

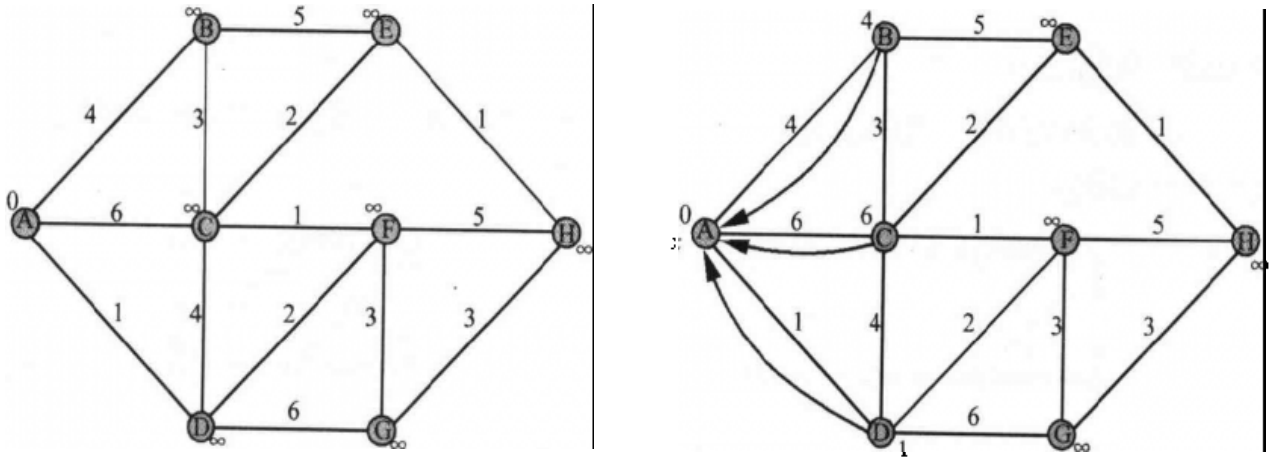


图 16.25 Dijkstra 算法的跟踪过程（1）

算法执行第一遍时，具有最高优先级的点时 x =A，因为该点的距离域为 0，其他域为无穷大。步骤 7～10 执行 3 次，分别针对 A 点的 3 个邻居：y=B，C，D。对这 3 个点计算

距离值 s，分别为 0+4=4，0+6=6，0+1=1。每一个值都小于这些点当前距离域的值（无穷大），所以，这 3 个值都分别赋给这 3 个点的距离域，这 3 个点的向后引用点设为 A。

算法执行第二遍时，优先级队列中最高优先级的点是 x=D，因为它的距离域等于 1。步骤 7～10 也执行 3 次，分别针对点 D 的 3 个未访问过的邻居 y=C，F，G。对这 3 个点计算距离值 s，分别为 1+4=5，1+2=3，1+6=7。每一个值都小于这些点当前距离域的值，于是这 3 个值都分别赋给这 3 个点的距离域，且向后引用点设为 D。注意这时点 C 的距离域的值发生了变化。

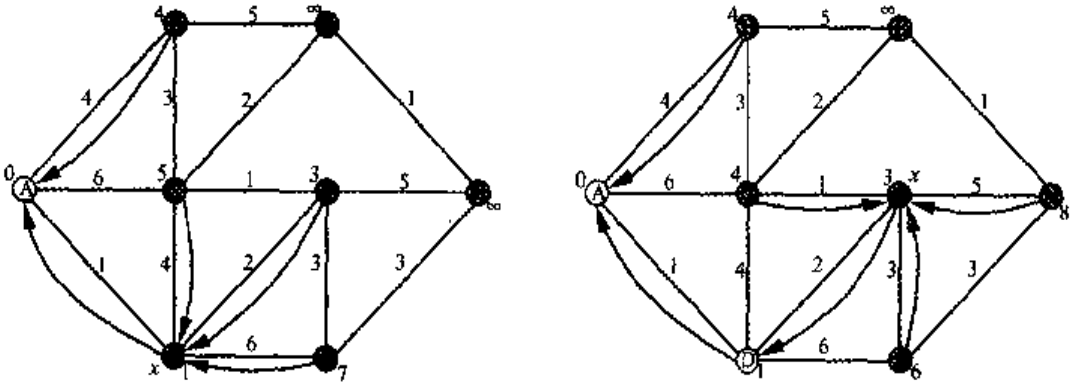


图 16.26　Dijkstra 算法的跟踪过程（2）

算法执行第三遍时，优先级队列中最高优先级的点是 x=F，因为它的距离域等于 3。步骤 7～10 也执行 3 次，分别针对点 F 的 3 个未访问过的邻居 y=C，G，H。对这 3 个点计算距离值 s，分别为 3+1=4，3+3=6，3+5=8。每一个值都小于这些点当前距离域的值，于是这 3 个值都分别赋给这 3 个点的距离域，且向后引用点设为 F。注意这时点 C 的距离域的值又发生了变化。

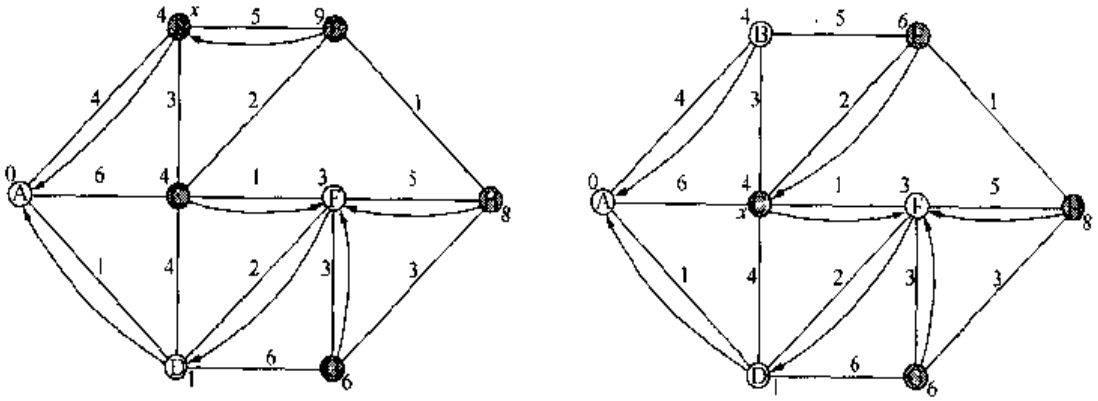


图 16.27　Dijkstra 算法的跟踪过程（3）

算法执行第四遍时，优先级队列中最高优先级的点是 x=B，因为它的距离域等于 4。步骤 7～10 执行二次，这时 y=C，E。对这两个点计算距离值 s，分别为 4+3=7，4+5=9，对于点 E，9 比它当前距离域的值（无穷大）小，所以距离域的值赋为 9，它的向后引用点设为点 B。然而，s 为 7 不比点 C 的当前距离域的值小，所以该域不发生变化。

标准分享网 www.bzfxw.com 免费下载

算法执行跌代过程，依次为 x=C，E，G，最后为 H。

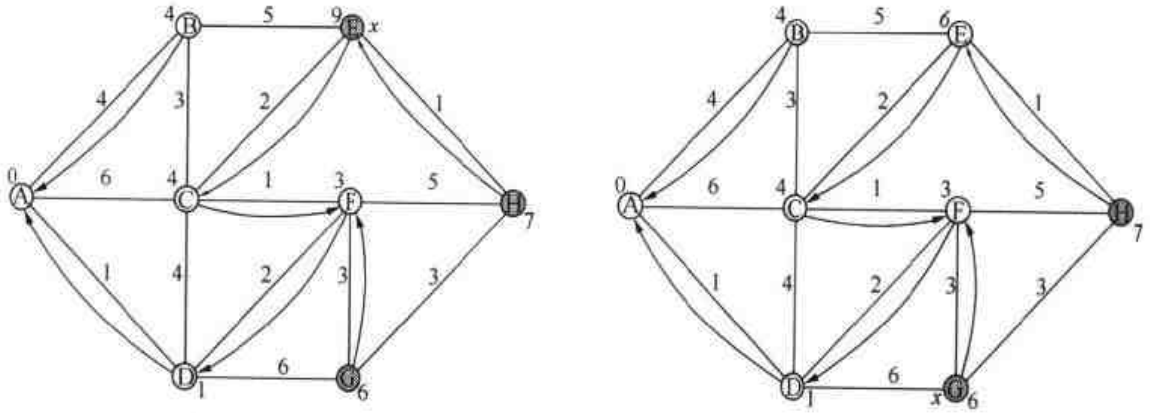


图 16.28 Dijkstra 算法的跟踪过程（4）

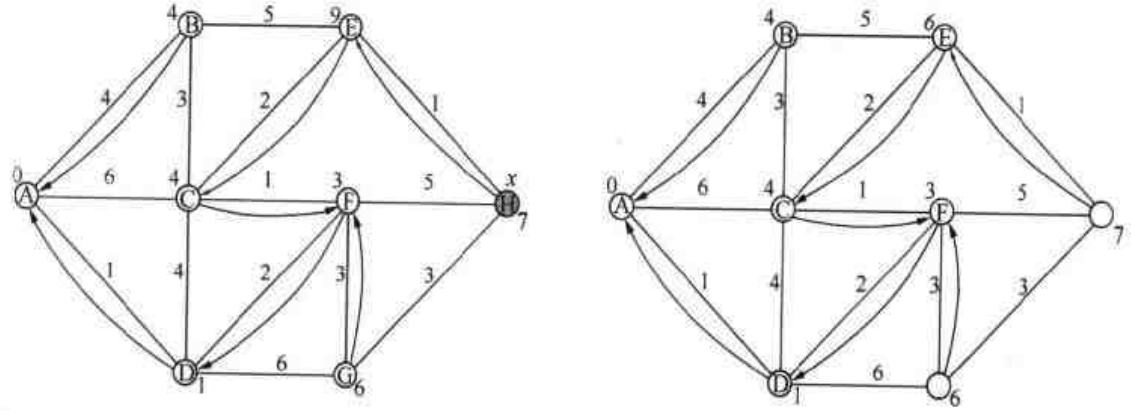


图 16.29 Dijkstra 算法的跟踪过程（5）

算法执行完后，形成结果。例如，从点 A 到点 E 的最短路径是 ADFCE，长度为 6。

例 16.25 Dijkstra 算法的 Java 实现

下面是算法 16.1 的 Java 实现。它定义了类 Network，Network 类的实例表示了加权有向图。

```
public class Network
{  Vertex start;

   private class Vertex
   {  Object object;
   Edge edges;
   Vertex nextVertex;
   boolean done;
   int dist;
   Vertex back;
}
```

```
    private class Edge
    {  Vertex to;
       int weight;
       Edge nextEdge;
    }
    public Network()
    {  if (start != null)
       {  start.dist = 0;
          for (Vertex p = start.nextVertex; p != null; p = p.nextVertex)
            p.dist = Integer.MAX_VALUE; // infinity
       }
    }

    public void findshortestPaths()
    {  // implements Dijkstra's Algorithm:
       for (Vertex v = start; v !=null; v = closestVertex())
       {  for (Edge e = v.edges; e != null; e = e.nextEdge)
          {  vertex w = e.to;
             if (!w.done && v.dist+e.weight <w.dist)
             {  w.dist = v.dist+e.weight;
                w.back = v;
             }
          }
          v.done = true;
       }
    }
    private Vertex closestVertex()
    {  // returns the vertex with minimal dist among those not done:
       Vertex v = null;
       int minDist = Integer.MAX_VALUE;
       for (Vertex w = start; w != null; w = nextVertex)
         if (!w.done && w.dist <minDist) {  v = w; minDist = w.dist;  }
       return v;
    }
  }
```

在实现过程中，我们使用了一个简单的搜索方法 closestVertex () 来代替一个优先级队列。这种方法效率略低，需要运行 $O(n)$时间，而优先级队列的运行时间为 $O(\lg n)$。

16.13 图的遍历算法

由 Dijkstra 算法生成的路径是图的一个最小生成树。对于该图来说，它的总加权路径之和最小；也就是说，再没有其他生成树的总加权路径之和比这小。该生成树是以宽度优先的方式生

标准分享网 www.bzfxw.com 免费下载

成的，因为每一次跌代过程是访问当前点的邻居点。这是遍历一个图的常用两种方法中的一种。

算法 16.2 宽度优先搜索算法（BFS）

（前提：$G=(V,E)$ 是一个图或有向图且初始节点为 v_0；每一个点有一个布尔变量表示的访问域，初始化为假 false；T 是边的集合，初始为空；L 是点的列表，初始为空。）

（结果：L 按 BFS 列出所有点，T 是图 G 的一个 BFS 生成树）

1. 初始化一个空队列 Q，用于临时存放点。
2. 将 v_0 压入队列 Q。
3. 若 Q 非空，重复执行步骤 4～6。
4. 从队列中弹出一个点到 x。
5. 将 x 加入 L 中。
6. 对于点 x 的每个邻居点 y 重复执行步骤 7。
7. 若 y 未被访问，执行步骤 8～9。
8. 将边 xy 加入 T 中。
9. 将点 y 加入队列 Q。

例 16.26 跟踪 BFS 算法

表 16.1 是算法 16.2 对图 16.30 的执行情况。

初始点 v_0=A。

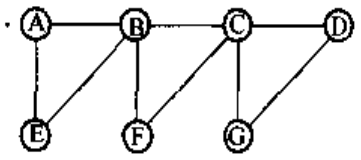


图 16.30 例 16.26 图

表 16.1 跟踪 BFS 算法

Q	*x*	*L*	*y*	*T*
A	A	A	B	AB
B			E	AB，AE
B，E	B		C	AB，AE，BC
E，C		A，B	F	AB，AE，BC，BF
E，C，F	E	A，B，E		
C，F	C	A，B，E，C	D	AB，AE，BC，BF，CD
F，D			G	AB，AE，BC，BF，CD，CG
F，D，G	F	A，B，E，C，F		
D，G	D	A，B，E，C，F，D		
G	G	A，B，E，C，F，D，G		

BFS 点的访问序列的结果返回在列 L 中，L=(A,B,E,C,F,D,G)，BFS 生成树的结果返回在集合 T 中，T={AB,AE,BC,BF,CD,CG}，如图 16.31 所示。

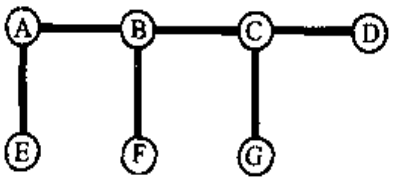


图 16.31 例 16.26 生成树

深度优先算法使用一个栈，而不是一个队列。

算法 16.3 深度优先算法（DFS）

（前提：$G=(V,E)$是一个图或一个有向图，并有初始点 v_0，每一个点有一个布尔变量表示的访问域，初始化为假 false；T 是边的集合，初始为空；L 是点的列表，初始为空。）

（结果：L 按 DFS 列出所有点，T 是图 G 的一个 DFS 生成树）

1. 初始化一个空栈 S，用于临时存储点。
2. 将 v_0 加入 L 中。
3. 将 v_0 压入栈 S。
4. 标识 v_0 被访问过。
5. 若 S 非空，重复步骤 6～8。
6. 约定 x 为栈 S 的首元素。
7. x 有未被访问的邻居点，作步骤 9～13。
8. 否则，从栈 S 中弹出，并返回步骤 5。
9. 约定 y 是与 x 邻居的未被访问的点。
10. 将边 xy 加入 T。
11. 将 y 加入 L。
12. 将 y 压入 L。
13. 标识 y 被访问过。

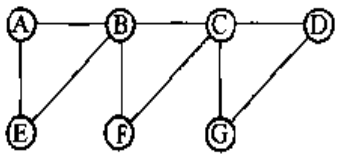


图 16.32　例 16.27 图

例 16.27　跟踪算法 DFS

表 16.2 是算法 16.3 对图 16.32 的执行结果。初始点 v_0=A。

表 16.2　跟踪 DFS 算法

L	S	x	y	T
A	A	A	B	AB
A, B	A, B	B	C	AB, AC
A, B, C	A, B, C	C	D	AB, BC, CD
A, B, C, D	A, B, C, D	D	G	AB, BC, CD, DG
A, B, C, D, G	A, B, C, D, G	G		
	A, B, C, D	D	D	
	A, B, C	C	F	AB, BC, CD, DG, CF
A, B, C, D, G, F	A, B, C, F	F		
	A, B, C	C		
	A, B	B	E	AB, BC, CD, DG, CF, BE
A, B, C, D, G, F, E	A, B, E	E		
	A, B	B		
	A	A		

DFS 点的访问序列的结果返回在列 L 中，L=(A,B,C,D,G,F,E)，DFS 生成树的结果返回在集合 T 中，T={AB,BC,CD,DG,CF,BE}，如图 16.33 所示。

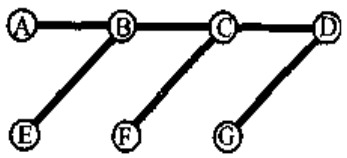


图 16.33　例 16.27 生成树

由于深度优先算法使用栈，所以它自然有相应的递规算法。

标准分享网 www.bzfxw.com 免费下载

算法 16.4 DFS 的递归算法

（前提：$G=(V,E)$是一个图或一个有向图，并有初始点 x，每一个点有一个布尔变量表示的访问域，初始化为假 false；T 是全局边的集合；L 是全局点的列表。）

（结果：L 按 DFS 列出所有点，T 是图 G 的一个 DFS 生成树）

1. 标识 x 被访问过。
2. 将 x 加入 L 中。
3. 对点 x 的每一个未被访问的邻居点 y 重复步骤 4～5。
4. 将边 xy 加入 T 中。
5. 对以初始点为 y 的子图执行 DFS 算法。

例 16.28 跟踪递归 DFS 算法

表 16.3 算法 16.4 对图 16.34 的执行结果。初始点 v_0=A。

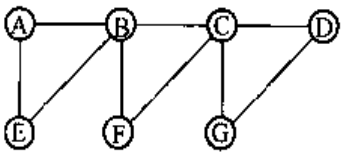


图 16.34 例 16.28 图

表 16.3 跟踪递归 DFS 算法

L	*x*	*Y*	*T*
A	A	B	AB
A, B	B	C	AB, BC
A, B, C	C	D	AB, BC, CD
A, B, C, D	D	G	AB, BC, CD, DG
A, B, C, D, G	G		
	D		
	C	F	AB, BC, CD, DG, CF
A, B, C, D, G, F	F		
	C		
	B	E	AB, BC, CD, DG, CF, BE
A, B, C, D, G, F, E	E		
	B		
	A		

结果当然与例 16.27 的一样。仅有的差别是用系统栈来代替显示的栈 S。

例 16.29 图遍历算法的实现

这示例 16.25 介绍的用 Network 类实现两个遍历算法：

```
public class Network
{  Vertex start;

   private class Vertex
```

```
{  Object object;
   Edge edges;
   Vertex nextVertex;
   boolean visited;
}
private class Edge
{  Vertex to;
   int weight;
   Edge nextEdge;
}

public static void visit(Vertex x)
{  System.out.println(x.object);
}

public void breadthFirstSearch()
{  if (start = = null) return;
   Vector queue = new Vector();
   Visit(start);
   Start.visited = true;
   queue.addElement(start);
   While (!queue.isEmpty())
   {  Vertex v = queue.firstElement();
      queue.removeElementAt(0);
      for (Edge e = v.edges; e != null; e = e.nextEdge)
      {  Vertex w = e.to;
         if (!w.visited)
         {  visit(w);
            w.visited = true;
            queue.addElement(w);
         }
      }
   }
}
public void depthFirstSearch()
{  if (start != null) depthFirstSearch(start);
}

public void depthFirstSearch(Vertex x)
{  visit(x);
   x.visited = true;
   for (Edge e = x.edges; e != null; e = e.nextEdge)
   {  Vertex w = e.to;
      if (!w.visited) depthFirstSearch(w);
```

标准分享网 www.bzfxw.com 免费下载

```
        }
    }
}
```

这里使用的是递归深度优先搜索算法。这需要 `depthFirstSearch()`方法以零个参数的形式来启动该递归算法。

复 习 题

16.1 一个图与一个简单图之间有什么区别?

16.2 在一个无向图中，一条边自己可以是一条路径吗?

16.3 连接点和相邻点之间有什么区别?

16.4 根据同构图的定义，证明两图同构容易，还是证明两图不同构容易，为什么?

16.5 定理 16.4 的 5 个条件是两图同构的充分条件吗?

16.6 为什么一个简单图的定义禁止环，而有向图的定义允许环?

16.7 判定真或假：

a. 若一个图有 n 个点，$n(n-1)/2$ 条边，则它一定是一个完全图。

b. 一条路径的长度一定小于图的大小。

c. 一条回路的长度等于它包含的不同点的个数。

d. 如果一个图的关联矩阵有 n 条边，$n(n-1)/2$ 列，则该图一定是一个完全图。

e. 在一个有向图的关联矩阵中，每一行的项数的总和等于该点的入度。

f. 一个图的关联矩阵的所有项的和等于 $2|E|$。

g. 一个有向图的关联矩阵的所有项的和等于 0。

16.8 一个图(V,E)若$|E|=\Theta(|V|^2)$，称为稠密；若$|E|=O(|V|)$，称为稀疏的。

a. 对稠密图来说，这 3 种表示（邻接矩阵、关联矩阵、邻接表）哪一种最好?

b. 对稀疏图来说，这 3 种表示（邻接矩阵、关联矩阵、邻接表）哪一种最好?

习 题

16.1 对图 16.35，求下述值：

a. 大小 n。

b. 点集合 V。

c. 边集合 E。

d. 每个点 x 的度数 $d(x)$。

e. 长度为 3 的路径。

f. 长度为 5 的路径。

g. 长度为 4 的回路。

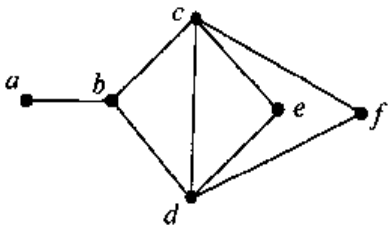


图 16.35 习题 16.1 图

h. 一个生成树。

i. 邻接矩阵。

j. 关联矩阵。

k. 邻接表。

16.2　对图 16.36 所示的有向图，求下述值：

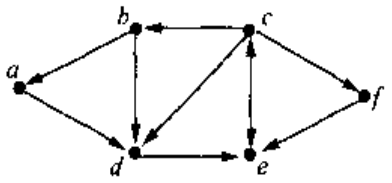


图 16.36　习题 16.2 图

a. 大小 *n*。

b. 点集合 *V*。

c. 边集合 *E*。

d. 每个点 *x* 的入度数 *id*(*x*)。

e. 每个点 x 的出度数 *od*(*x*)。

f. 长度为 3 的路径。

g. 长度为 5 的路径。

h. 长度为 4 的回路。

i. 邻接矩阵。

j. 关联矩阵。

k. 邻接表。

16.3　画出 *n* 个点的完全图，*n*=2，3，4，5，6。

16.4　判定图 G_1 是欧拉图还是哈密顿图，如图 16.37 所示 。

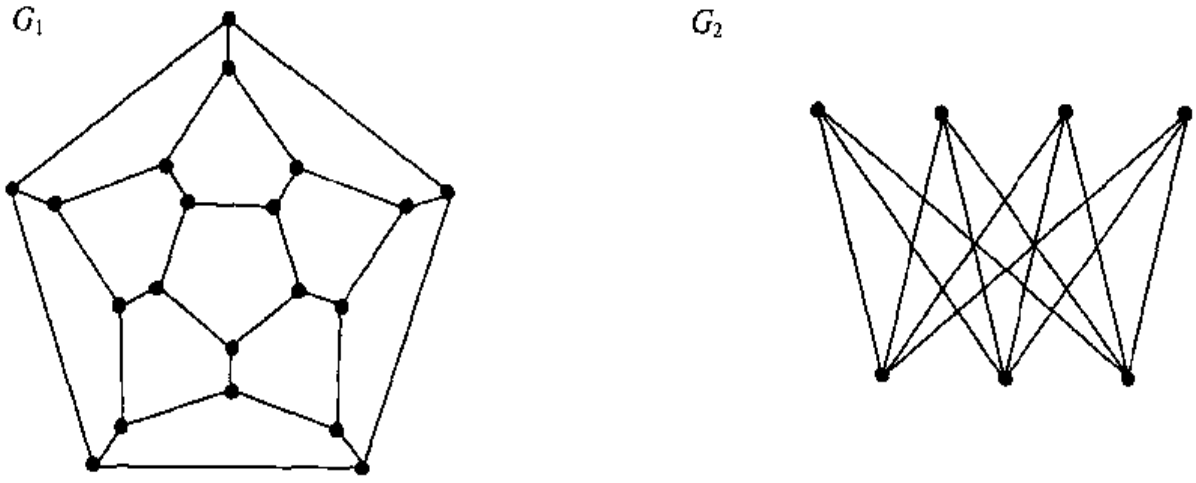


图 16.37　欧拉和哈密顿图

16.5　判定图 G_2 是欧拉图还是哈密顿图，如图 16.37 所示。

16.6　例 16.7 图的各个子图（如图 16.38 所示），判定它们是连通图、无环图还是生成树。

16.7　找出两个图非同构，但符合定理 16.4 的 5 个条件。

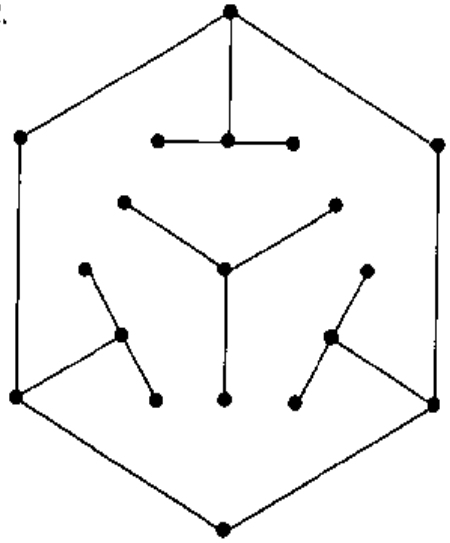
16.8　证明推论 16.2。

16.9　描述 *n* 个点的完全图的邻接矩阵。

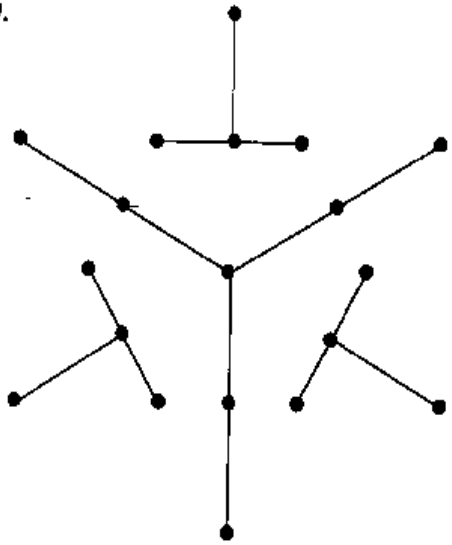
16.10　描述 *n* 个点的完全图的关联矩阵。

标准分享网 www.bzfxw.com 免费下载

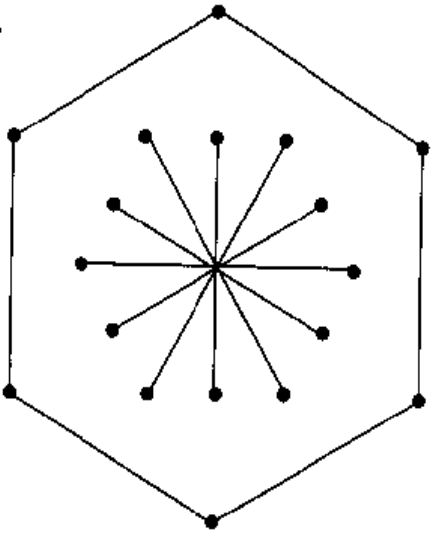
a.



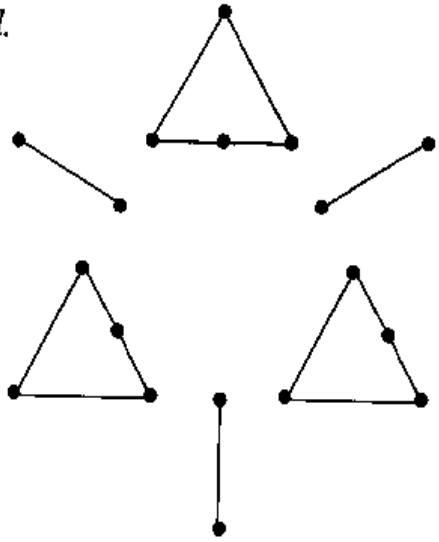
b.



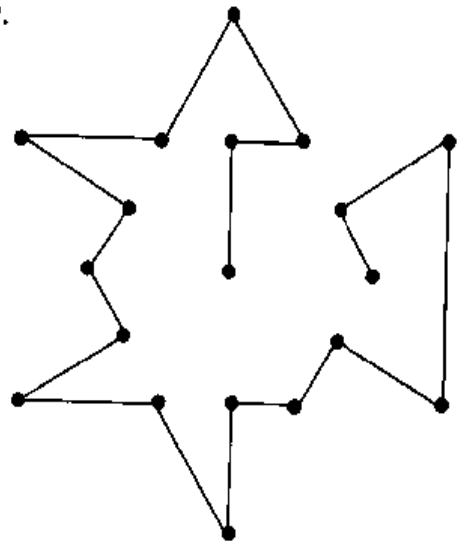
c.



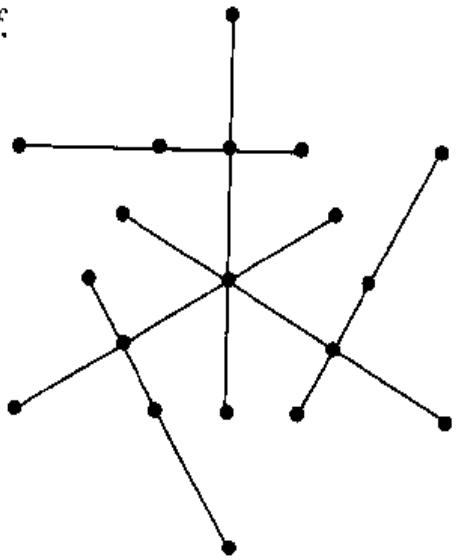
d.



e.



f.



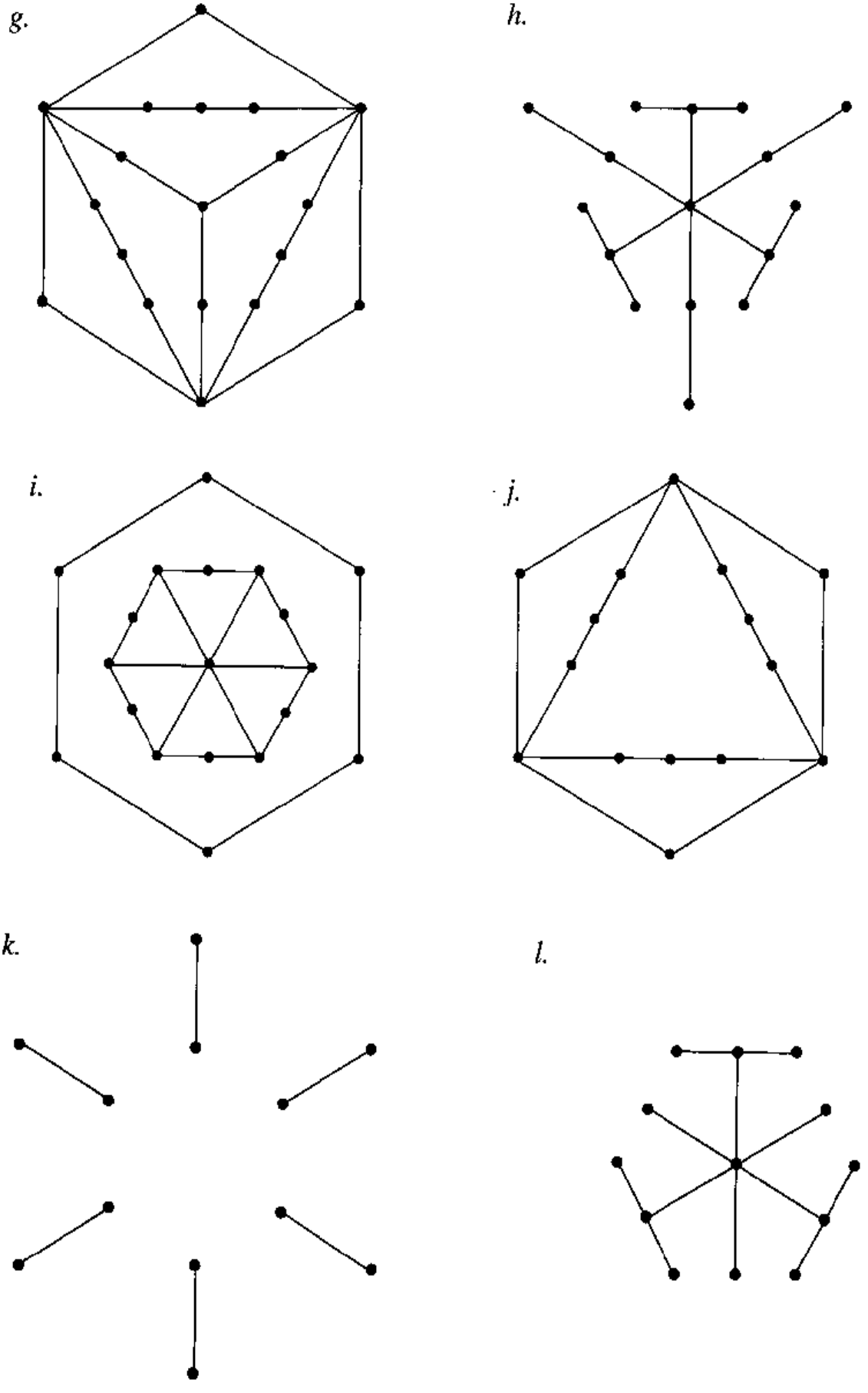


图 16.38　习题 16.7 图（a～l）

标准分享网 www.bzfxw.com 免费下载

16.11 假定 G_1 是该邻接表（见图 16.39）表示的图：

a. 画出 G_1。

b. G_1 是有向图吗？

c. G_1 是强连通吗？

d. G_1 是弱连通吗？

e. G_1 是无环图吗？

f. 给出图 G_1 的邻接矩阵。

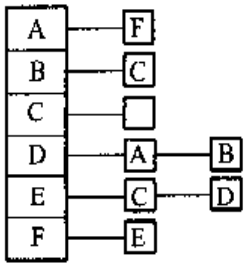


图 16.39 习题 16.11 图

16.12 假定 G_1 是该邻接矩阵（见图 16.40）表示的图：

a. 画出 G_1。

b. G_1 是有向图吗？

c. G_1 是强连通吗？

d. G_1 是弱连通吗？

e. G_1 是无环图吗？

f. 给出图 G_1 的邻接矩阵。

$$\begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 \end{bmatrix}$$

图 16.40 习题 16.12 图

16.13 根据图 16.41 的加权有向图 G_2，回答问题。

a. 画出该图的邻接矩阵。

b. 画出该图的邻接表。

c. 该图是连通的吗？

d. 该图是无环的吗？

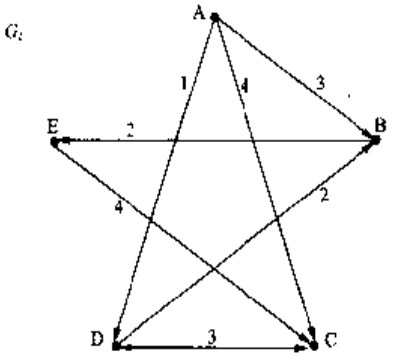
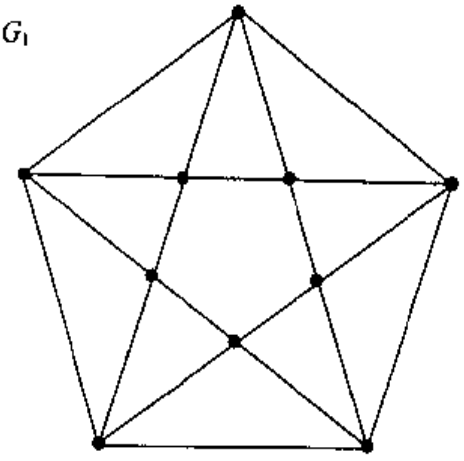


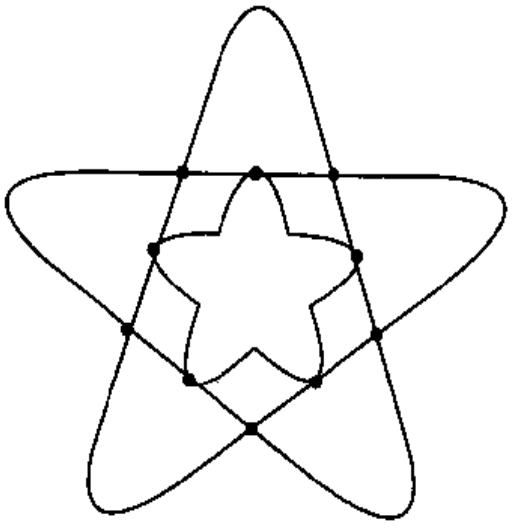
图 16.41 习题 16.13 图

16.14 判定图 16.42 中哪些图是同构的。注意这 7 个图的大小均为 10。

G_1



G_2



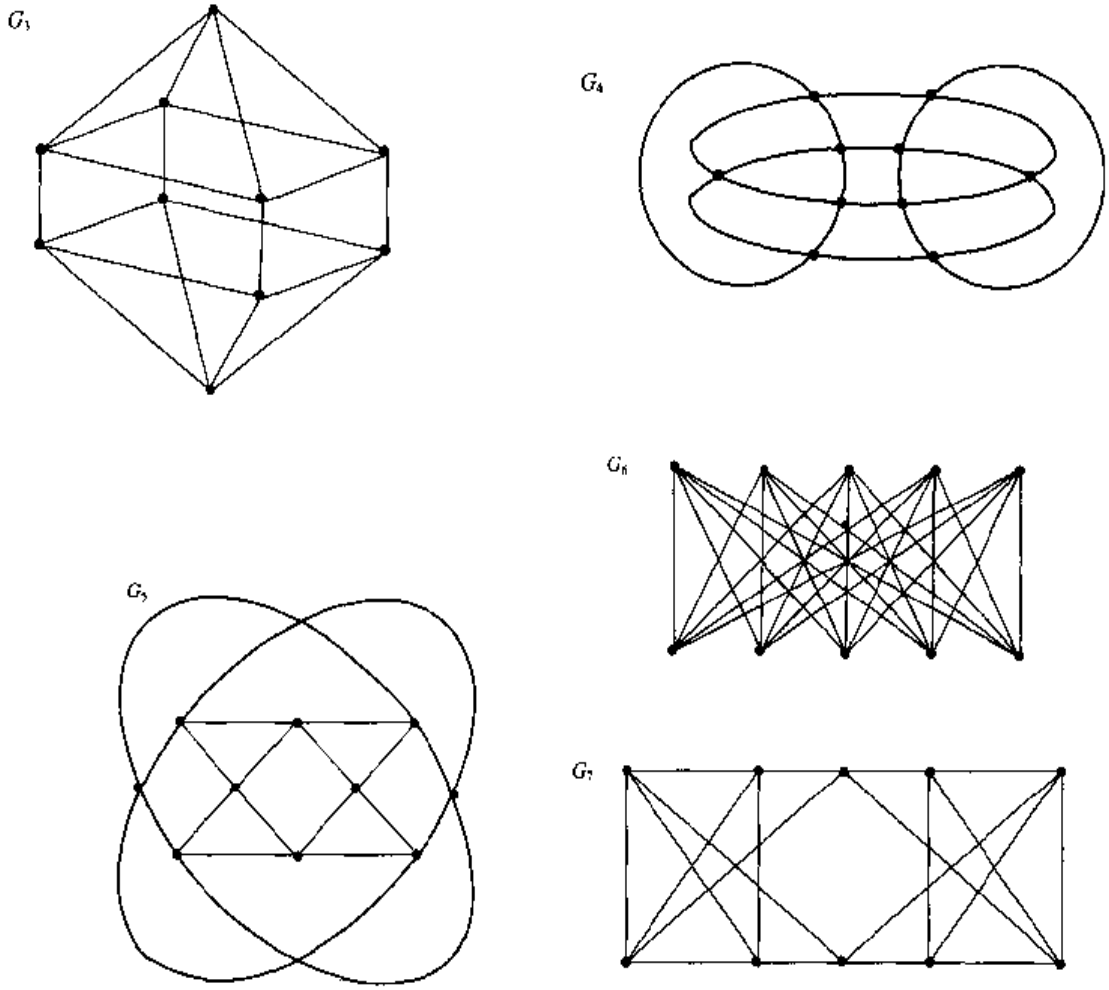


图 16.42　习题 16.14 图

16.15　G_1、G_2 如图 16.43 所示（是例 16.22 图）。

a. 判定是 G_1、G_2 同构吗？

b. 找出图 G_2 一条欧拉回路，或说明不存在欧拉回路。

C. 找出图 G_2 一条哈密顿回路，或说明不存在哈密顿回路。

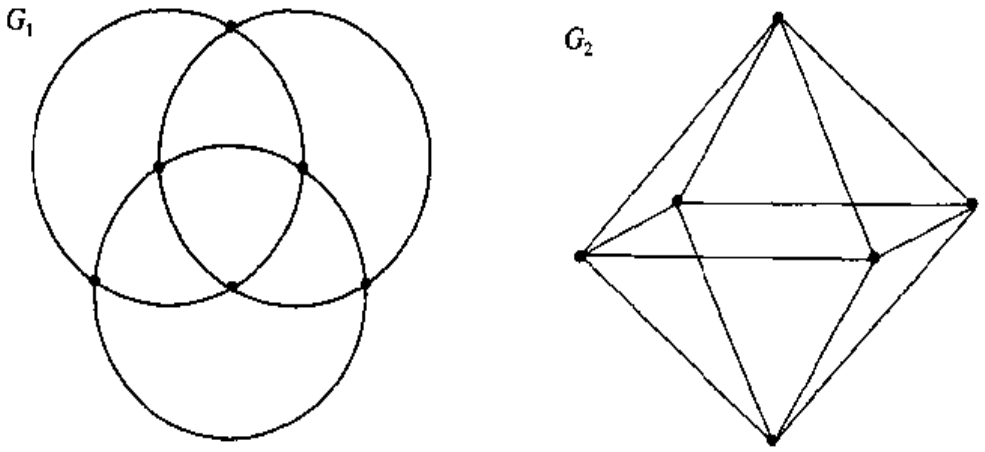


图 16.43　习题 16.15 图

标准分享网 www.bzfxw.com 免费下载

16.16 一个 n 点的轮状图是指大小为 $n+1$ 的图，由 n 个点的循环构成，且这 n 个点与中间一个单独点均邻接。例如，图 16.44 是 6 个点的轮状图。描述：

a. 一个 n 点的轮状图的邻接矩阵。

b. 一个 n 点的轮状图的关联矩阵。

c. 一个 n 点的轮状图的邻接表。

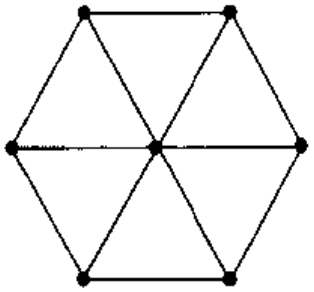


图 16.44 习题 16.16 图

16.17 用 Dijkstra 算法执行图 16.45 中的 G_8，描述具体执行情况，并求出最短路径和从点 A 到各点的距离。

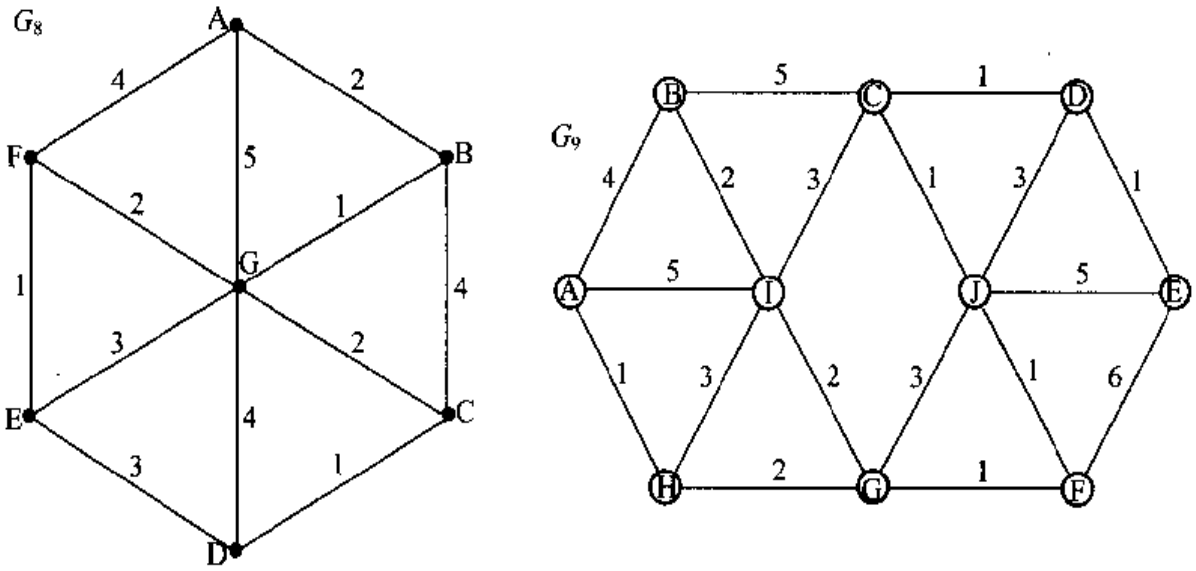


图 16.45 习题 16.17 及 16.18 图

16.18 用 Dijkstra 算法执行图 16.45 中的 G_9，描述具体执行情况，并求出最短路径和从点 A 到各点的距离。

16.19 有 4 种遍历二叉树的算法：前序遍历、中序遍历、后序遍历、同层次遍历。若一个二叉树是一个连通无环图，下面搜索方法属于哪个算法：

a. 深度优先搜索

b. 宽度优先搜索

16.20 用下列指定算法对图 16.46 进行遍历。给出访问点的顺序以及最后形成的生成树：

a. 宽度优先搜索。

b. 深度优先搜索，开始点为 A，打印出被访问的每一个点。

16.21 对于图 16.47 的加权图 G_1：

a. 画出邻接矩阵。

b. 画出关联矩阵。

c. 画出邻接表。

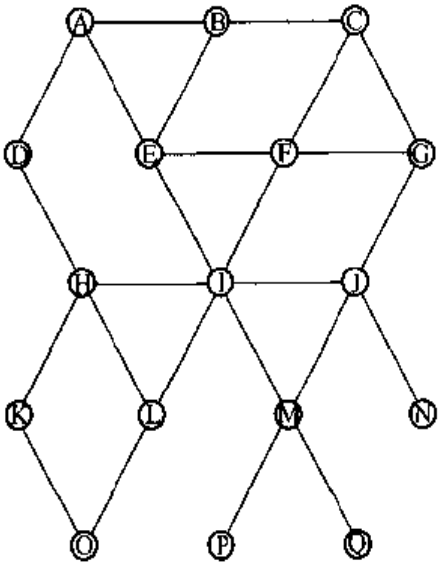


图 16.47　习题 16.21 图

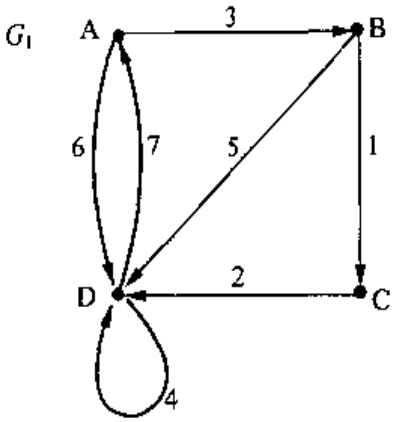


图 16.46　习题 16.20 图

复习题答案

16.1　没有环路和重复边的图是简单图。

16.2　不是：在一个非有向图中，一条边不可能是一条路径，因为一条边是两元素的集合（非有序对），而一条路径是一个序列（一个有序的点的列表）。

16.3　如果从一点到另一点存在一条路径，那么它们是连接的；如果两点之间有一条边，那么，它们是相邻的。

16.4　仅根据同构图的定义，证明两图同构容易，因为只需要找出同构函数，并证明之。而根据定义，证明两图非同构需要证明 n!种映射关系都是非同构的。

16.5　不：这 5 个条件不是两图同构判定的充分条件。很有可能两图满足这 5 个条件，却是非同构图。（参见问题 16.7）

16.6　一个图的定义禁止回路的原因是在一个图中，一条边是两元素集合，需要两元素不同。在有向图的定义中，一条边是有序对，因此允许两元素相同。

16.7　*a*. 真。

b. 真。

c. 真。

d. 真。

e. 假。

f. 真。

16.8　邻接矩阵适合一个稠密图，因为它较紧凑，并且提供快速访问。邻接表是适合稀疏图，因为它允许很容易的插入和删除边。

标准分享网 www.bzfxw.com 免费下载

习 题 解 答

16.1 *a.* n=6。

b. $V = \{a,b,c,d,e,f\}$.

c. $E = \{ab,bc,bd,cd,ce,de,cf,df\}$

d. $d(a)=1$ $d(b)=3$ $d(e)=d(f)=2$,$d(c)=d(d)=4$.

e. 路径 *abcd* 的长度是 3。

f. 路径 *abcfde* 的长度是 5。

g. 回路 *bcedb* 的长度是 4。

h. 生成树如图 16.48 所示。

i. 邻接矩阵见图 16.49 左上。

j. 关联矩阵见图 16.49 左下。

k. 邻接表见图 16.49 右。

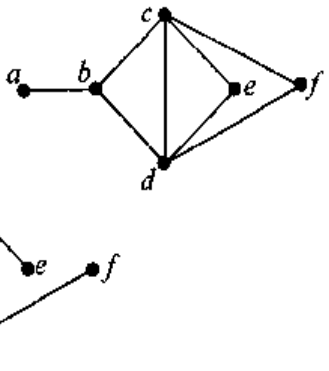


图 16.48 习题 16.1 答案

	a	b	c	d	e	f
a	F	T	F	F	F	F
b	F	F	T	T	F	F
c	F	T	F	T	T	T
d	F	T	T	F	T	T
e	F	F	T	T	F	F
f	F	F	T	T	F	F

a	1	0	0	0	0	0	0	0
b	1	1	1	0	0	0	0	0
c	0	1	0	1	1	1	0	0
d	0	0	1	1	0	0	1	1
e	0	0	0	0	1	0	1	0
f	0	0	0	0	0	1	0	1

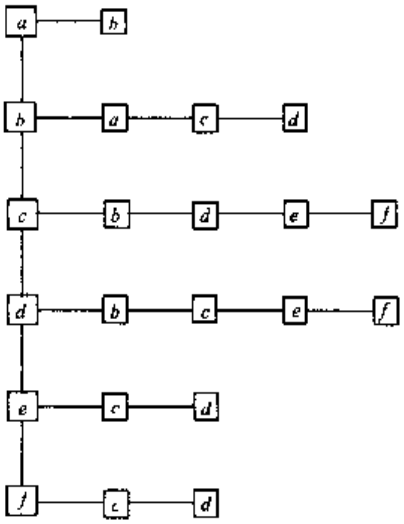


图 16.49 习题 16.1 答案

16.2 a. n=6。

b. $V = \{a,b,c,d,e,f\}$.

c. $E = \{ab,ba,bd,cb,cd,ce,cf,de,ec,fe\}$

d. $id(a)=id(b)=id(c)=id(f)=1$,$id(d)=id(e)=3$.

e. $od(a)=od(d)=od(e)=od(f)=1$,$od(b)=2$,$od(c)=4$.

f. 路径 adec 的长度是 3。

g. 路径 fecbad 的长度是 5。

h. 回路 adcba 的长度是 4。

i. 生成树见图 16.50。

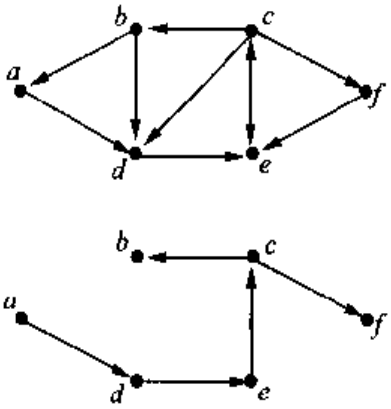


图 16.50 习题 16.2 答案

j. 邻接矩阵见图 16.51 左上。

k. 关联矩阵见图 16.51 左下。

l. 邻接表见图 16.51 右。

	a	*b*	*c*	*d*	*e*	*f*
a	F	F	F	T	F	F
b	T	F	F	T	F	F
c	F	T	F	T	T	T
d	F	F	F	F	T	F
e	F	F	T	F	F	F
f	F	F	F	F	T	F

a	1	-1	0	0	0	0	0	0	0	0
b	0	1	1	-1	0	0	0	0	0	0
c	0	0	0	1	1	1	1	0	-1	0
d	-1	0	-1	0	-1	0	0	1	0	0
e	0	0	0	0	0	-1	0	-1	-1	-1
f	0	0	0	0	0	0	-1	0	0	1

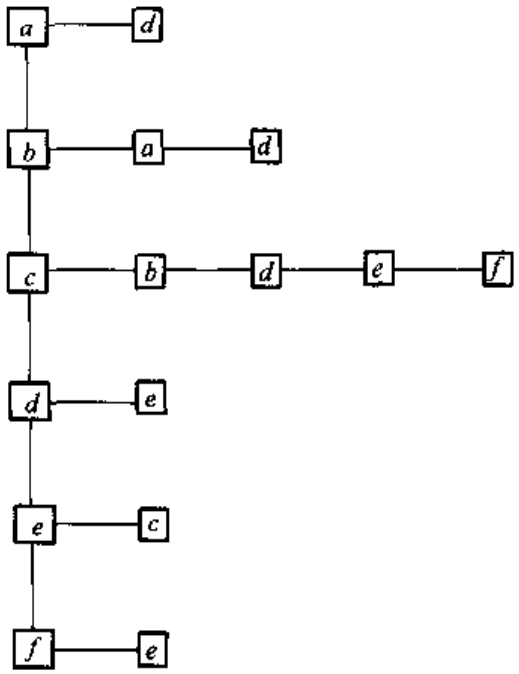


图 16.51　习题 16.2 答案

16.3　完全图（如图 16.52 所示）：

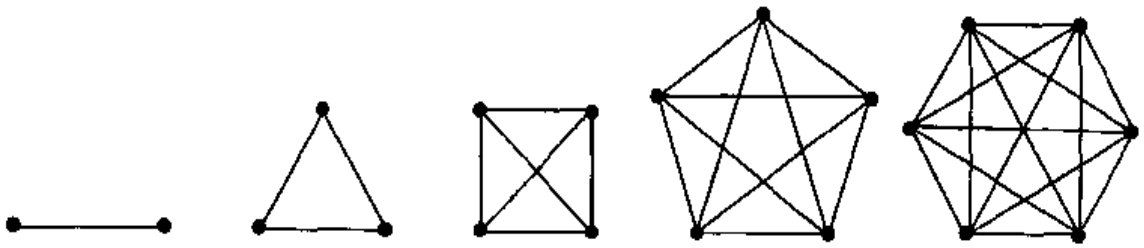


图 16.52　习题 16.3 答案

16.4　图 G_1 不是欧拉图，因为它有奇数点，但是它存在的哈密顿回路说明是哈密顿图（如图 16.53 所示）。

16.5　图 G_2 既不是欧拉图，也不是哈密顿图。

16.6　*a*. 非连通，循环且有生成树。

b. 非连通，非循环且有生成树。

c. 非连通，循环且有生成树。

d. 非连通，循环且无生成树。

e. 连通，非循环且有生成树。

f. 连通，非循环且有生成树。

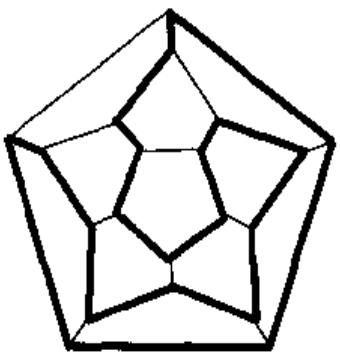


图 16.53　习题 16.4 答案

标准分享网 www.bzfxw.com 免费下载

g. 连通，循环且有生成树。

h. 非连通，非循环且无生成树。

i. 非连通，循环且有生成树。

j. 连通，循环且有无生成树。

k. 非连通，非循环且无生成树。

l. 非连通，非循环且无生成树。

16.7 图 16.54 满足定理 16.4 的 5 个条件，但它们不同构，因为左边的图有 4 个循环，包含度数为 2 的两个点，而右边的图却没有。

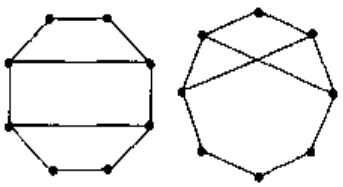


图 16.54 习题 16.7 答案

16.8 假定 G_1 与 G_2 同构，G 与 G_1 不同构，而 G 与 G_2 同构。根据定理 16.5 的第三部分，G 与 G_1 同构，这和前提 G 与 G_1 不同构矛盾，所以若 G_1 与 G_2 同构，G 与 G_1 不同构，则 G 与 G_2 不同构。

16.9 n 个点的完全图的邻接矩阵是一个布尔矩阵，矩阵的主对角线元素都为假，其他元素为真。

16.10 n 个点的完全图的关联矩阵 M_n 如图 16.55 所示：

它有 n 行、$n(n-1)/2$ 列（定理 16.2）。

若 n=2，它是 2 行 1 列矩阵，且每一项都为 1。

若 n>2，它是由矩阵 A 和矩阵 M_{n-1} 上面加上一行 false 水平连接起来。

M_2 =

1
1

M_3=

1	1	0
1	0	1
0	1	1

M_4 =

1	1	1	0	0	0
1	0	0	1	1	0
0	1	0	1	0	1
0	0	1	0	1	1

M_5 =

1	1	1	1	0	0	0	0	0	0
1	0	0	0	1	1	1	0	0	0
0	1	0	0	1	0	0	1	1	0
0	0	1	0	0	1	0	1	0	1
0	0	0	1	0	0	1	0	1	1

图 16.55 习题 16.10 答案

16.11 *a*. 有向图 G_1 如图 16.56 所示。

b. 这是一个有向图，它至少有一条有向边。

c. 该有向图不是强连通，从 C 到 D 不存在路径。

d. 该有向图是弱连通：它的嵌入图是连通的。

e. 该有向图不是非循环：它有回路 AFEDA。

f. 它的邻接矩阵如图 16.57 所示。

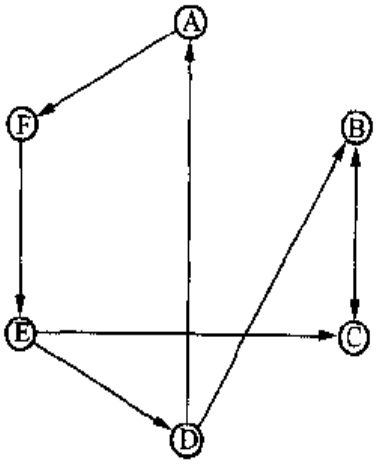


图 16.56 习题 16.11(a)答案

16.12 *a*. 有向图 G_1 如图 16.58 所示。

b. 不，这不是一个简单图，因为它存在环路。

c. 是，这是一个右向图：它的邻接矩阵是非对称的。

d. 是，该有向图是强连通的。

e. 是，该有向图是弱连通的。

f. 不，该有向图不是非循环的，它包含回路 ADBA。

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

图 16.57　习题 16.11(f)答案

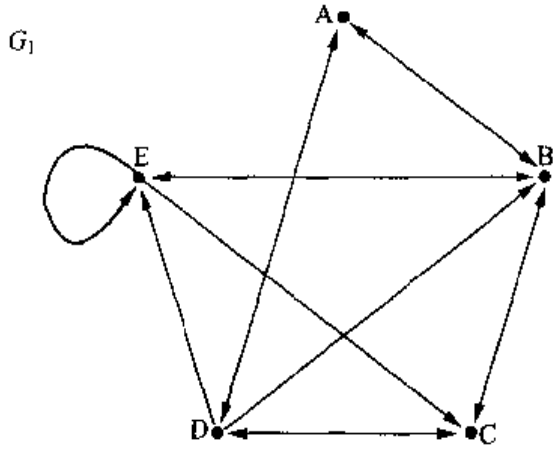


图 16.58　习题 16.12 答案

16.13　对于所给图，它的邻接矩阵和邻接表分别如图 16.59 所示：

$$\begin{bmatrix} \infty & 3 & 4 & 1 & \infty \\ \infty & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & 3 & \infty \\ \infty & 2 & 3 & \infty & \infty \\ \infty & \infty & 4 & \infty & \infty \end{bmatrix}$$

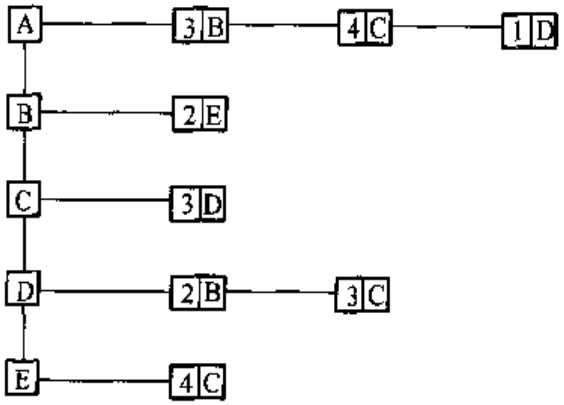
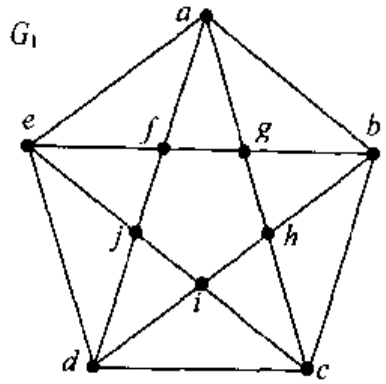
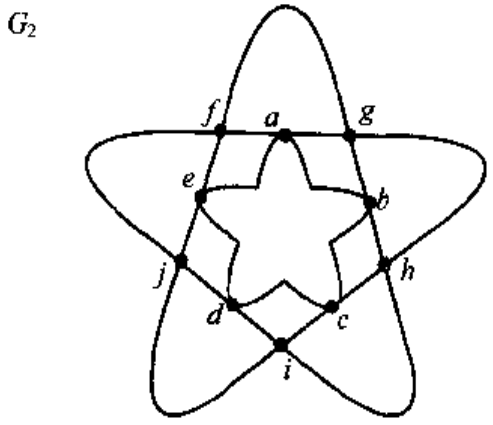


图 16.59　习题 16.13 答案

该图是非连通的，因为从 B 到 A 不存在路径。该图不是非循环，因为它包括回路 BECDB。

16.14　如图 16.60 这 7 个图中：





标准分享网 www.bzfxw.com 免费下载

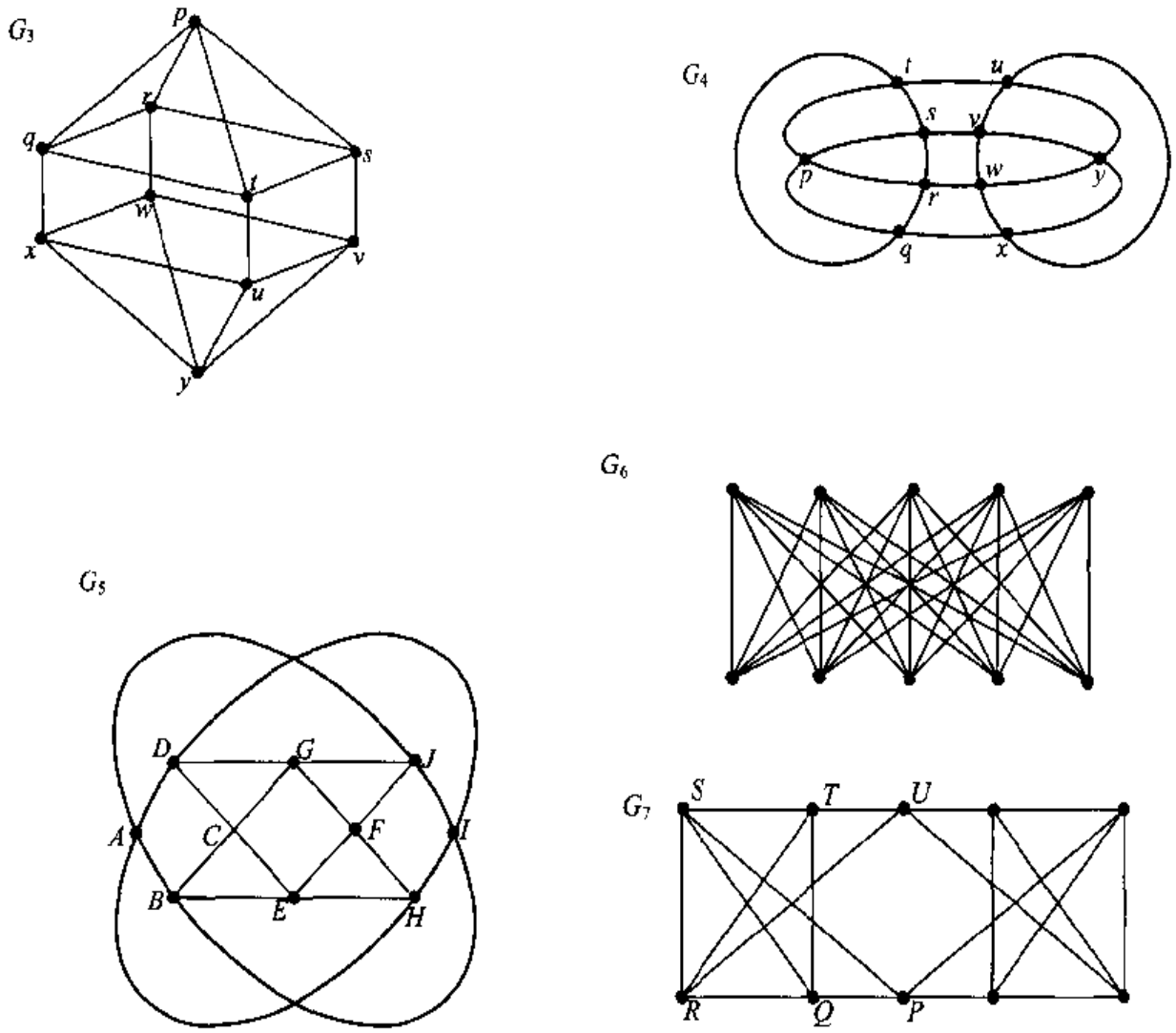


图 16.60 习题 16.14 答案

G_1 与 G_2 同构：通过点标号 a～j 的映射关系得到同构函数。

G_3 与 G_4 同构：通过点标号 p～y 的映射关系得到同构函数。

G_6 与其他图都不同构，因为它有 25 条边，而其他图都有 20 条边。

G_3（同样对于 G_4）与其他图都不同构，因为它有 4 个相邻的 3 点回路组成的一个金字塔（*pqr,prs,pst,ptq*），而其他图都没有（G_6 除外）。

G_5 与其他图都不同构，因为它有由 3 个相邻的 4 点回路组成的链（*ABCD,CEFG,FHIJ*），而其他图都没有（G_6 除外）。

G_7 同样与其他图都不同构，因为它有由 4 个相邻的 3 点回路组成的链 *PQS,QSR,SRT,RTU*），而其他图都没有（G_6 除外）。

16.15 a. 图 16.61 中的两个图是同构图，同构映射关系通过下面点标号定义出：

b. G_2 有欧拉回路，它是 ABCDEBFCADFEA。

c. G_2 有哈密顿回路，它是 ABCDFEA。

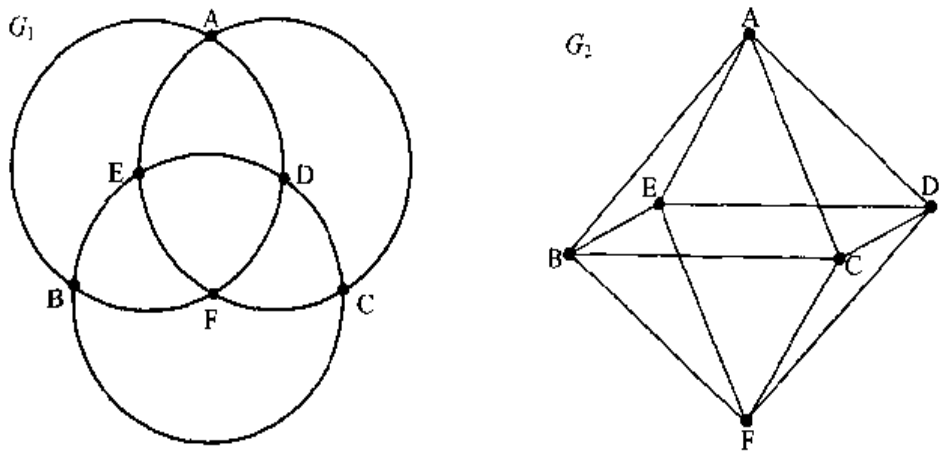
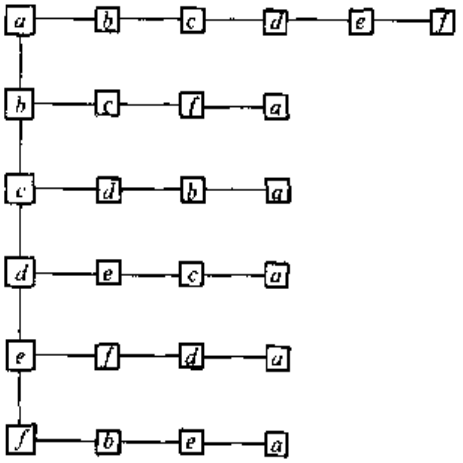


图 16.61　习题 16.15(a)答案

16.16　*a*. 轮状图的邻接矩阵的样子类似图 16.62 的矩阵 *A*。

b. 轮状图的关联矩阵的样子类似图 16.62 的矩阵 *B*（*n*=4）。通常，在第一行，有 *n* 个 1，其后跟着 *n* 个 0，接着下面是标准矩阵（主对角线为 1，其他为 0），后面跟着方阵（主对角线、从对角线为 1）。将这与习题 16.10 的递归方法进行比较。

c. 轮状图的邻接表的样子如图 16.62 所示。第一个点（中央点）的边列表有 *n* 条边，每一个对应一个点。其他边表有 3 个元素，其中一个指向中央点，其他指向各自的邻居点。



$A =$

F	T	T	T	T	T
T	F	T	F	F	F
T	T	F	T	F	F
T	F	T	F	T	F
T	F	F	T	F	T
T	F	F	F	T	F

$B =$

1	1	1	1	0	0	0	0
1	0	0	0	1	0	0	1
0	1	0	0	1	1	0	0
0	0	1	0	0	1	1	0
0	0	0	0	0	0	1	1

图 16.62　习题 16.16 答案

16.17　Dijkstra 算法对图 G_8 的执行情况如图 16.63 所示：

16.18　Dijkstra 算法对图 G_9 的执行情况如上所示。

16.19　*a*. 如果深度优先算法用于一个树，它属于前序遍历。

b. 如果宽度优先算法用于一个树，它属于同层序遍历。

16.20　*a*. 宽度优先算法访问点的顺序是 **ABDECHFIGKLJMONPQ**，它的生成树图 16.64 左图所示。

标准分享网 www.bzfxw.com 免费下载

b. 深度优先算法访问点的顺序是 **ABCFEIHDKLMJGN**，它的生成树图 16.64 右图所示。

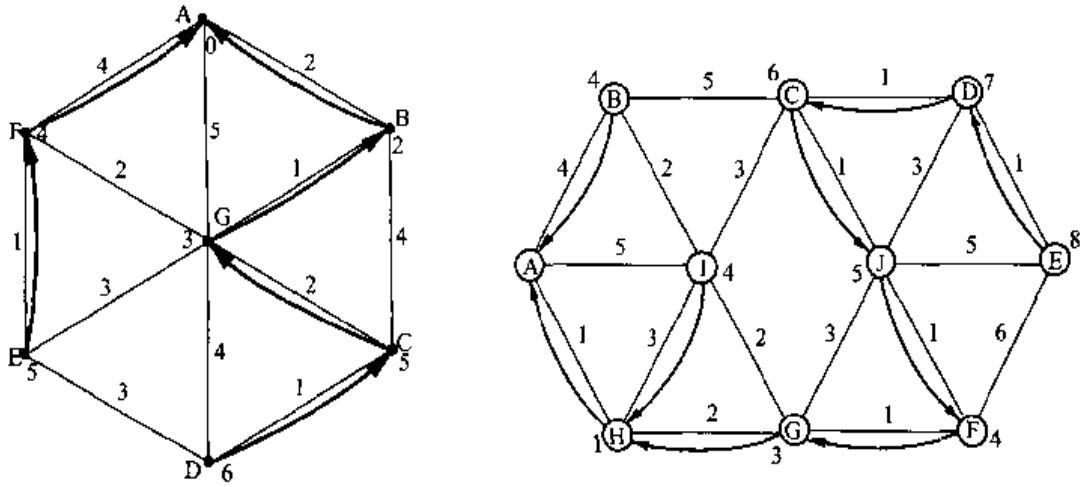


图 16.63 习题 16.17 答案

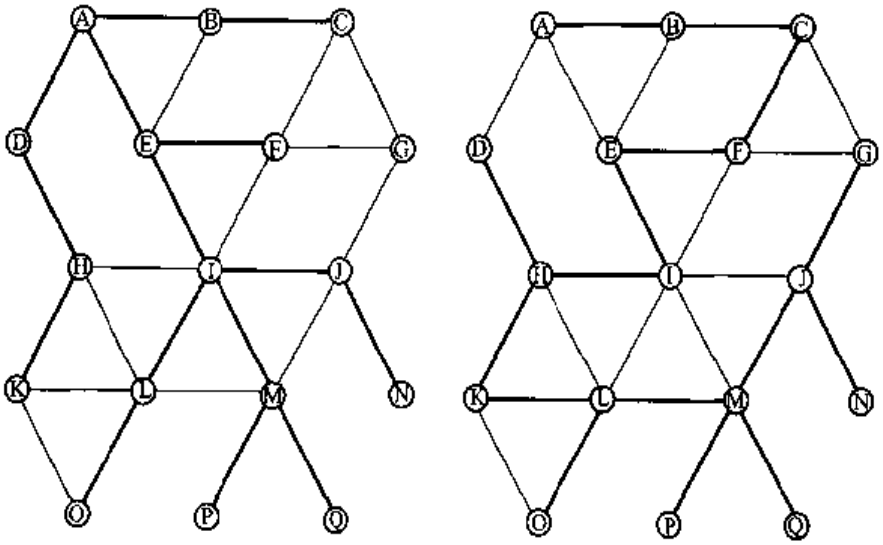


图 16.64 习题 16.20 答案

16.21 该图的邻接矩阵和邻接表分别如图 16.65 中左图、右图所示：

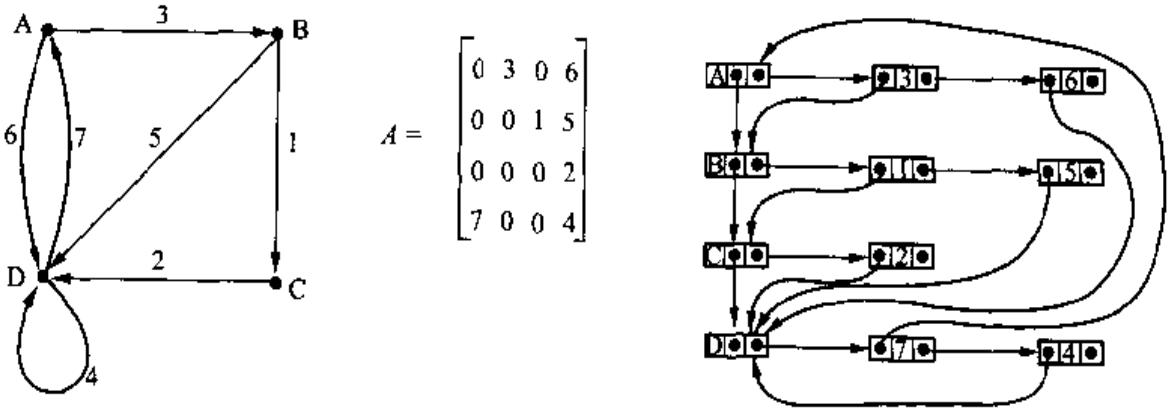


图 16.65 习题 16.21 答案

附录 A　基础数学知识

作为一门科学，计算机科学依赖于一些基本的理论原理，它们是数学中的理论推导出来或者是应用数学知识得来的。本附录概括了学习数据结构所需的数学知识。

A.1　下取整与上取整函数

下取整与上取整函数返回一个与给定实数最接近的整数。下取整函数用符号$\lfloor x \rfloor$标识，它表示不超过 x 的最大整数。上取整函数用符号$\lceil x \rceil$标识，它表示不小于 x 的最小整数。用数轴表示，如图 A.1 所示，就是下取整函数是在 x 左侧离 x 最近的整数，上取整函数是在 x 右侧离 x 最近的整数。

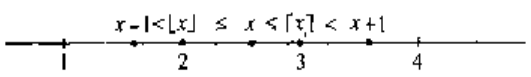


图 A.1　上取整与下取整

例 A.1　下取整与上取整函数

若 $x=2.718\,18$，则$\lfloor x \rfloor=2$，$\lceil x \rceil=3$

用 Z 表示所有整数的集合，R 表示所有实数的集合。则下取整函数与上取整函数是从 R 到 Z 的映射，也就是说每输入一个小数返回一个整数。在下述的讨论中，我们用 N 表示所有自然数的集合：$\mathbf{N}=\{n \in \mathbf{Z} \mid n \geq 1\}=\{1,2,3.....\}$。

定理 A.1

对于所有实数 x，下取整函数与上取整函数有如下性质：

a. $\lfloor x \rfloor = \max\{ m \in \mathbf{Z} \mid m \leq x \}$ 且 $\lceil x \rceil = \min\{ n \in \mathbf{Z} \mid n \geq x \}$

b. $\lfloor x \rfloor \leq x < \lfloor x \rfloor + 1$，且 $\lceil x \rceil - 1 < x \leq \lceil x \rceil$

c. $x-1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x+1$

d. 若 $n \in \mathbf{Z}$ 且 $n \leq x < n+1$，则 $n = \lfloor x \rfloor$。若 $n \in \mathbf{Z}$ 且 $n-1 < x \leq n$，则 $n = \lceil x \rceil$

e. 若 $x \in \mathbf{Z}$，则 $\lfloor x \rfloor = x = \lceil x \rceil$

f. 若 $x \notin \mathbf{Z}$，则 $\lfloor x \rfloor < x < \lceil x \rceil$

g. $\lfloor -x \rfloor = -\lceil x \rceil$ 且 $\lceil -x \rceil = -\lfloor x \rfloor$

h. $\lfloor x+1 \rfloor = \lfloor x \rfloor + 1$ 且 $\lceil x+1 \rceil = \lceil x \rceil + 1$

证明参见习题 A.2。

标准分享网 www.bzfxw.com 免费下载

A.2 对数

X取底为b的对数的含义是b的某次幂等于x。例如，底为10的1 000的对数等于3，因为$10^3=1\,000$，记作$\log_{10}1\,000=3$。社会科学者通常使用以10为底的对数，用logx表示$\log_{10}x$，称为常用对数。自然科学者以及数学家们通常使用以$e=2.718\,281\,828\,459\,045$为底的对数，用$\ln x$表示$\log_e x$，称为自然对数。计算机科学者通常使用以2为底的对数，用lgx表示$\log_2 x$，称为以2为底的对数，或二元对数。

作为数学函数，对数是指数函数的反函数：

$$y=\log_b x \quad \Leftrightarrow \quad b^y=x$$

例如，$\log_2 256 = 8$，因为$2^8 = 256$。上述等式可作为以b为底的对数定义。下述是对数的性质，这些性质可直接根据上面对数的定义推出。

定理 A.2 对数运算规则

a. $\log_b(b^y)=y$

b. $b^{\log_b x}=x$

c. $\log_b uv=\log_b u+\log_b v$

d. $\log_b u/v=\log_b u-\log_b v$

e. $\log_b u^v=v\log_b u$

f. $\log_b x=(\log_c x)/(\log_c b)=(\log_b c)(\log_c x)$

例 A.2 运用对数的运算规则

$\log_2 256=\log_2(2^8)=8$

$\log_2 1000=(\log_{10}1000)/(\log_{10}2)=3/0.301\,03=9.966$

$\log_2 1\,000\,000\,000\,000=\log_2 1000^4=4(\log_2 1000)=4(9.966)=39.86$

定理 A.3

二进制对数有下述性质：

a. 若 $p\in\mathbf{Z}$且$2^p<n<2^{p+1}$，则 $p=\lfloor\lg n\rfloor$且$p+1=\lceil\lg n\rceil$

b. 若$n\in\mathbf{N}$且$\lceil\lg(n+1)\rceil=\lfloor\lg n\rfloor+1$

证明参见习题A.4。

整数$\lfloor\lg n\rfloor$称为n的整二进制对数 (integral binary logarithm) 。它表示n被2除直到1的次数。例如，1000的整二进制对数等于9。因为1000可被平分9次：500，250，125，62，31，15，7，3，1。

例 A.3　测试整二进制对数

```
public class ExA01
{ public static void main(String[] args)
  { System.out.println("iLg(1) = " + iLg(1));
    System.out.println("iLg(2) = " + iLg(2));
    System.out.println("iLg(3) = " + iLg(3));
    System.out.println("iLg(4) = " + iLg(4));
    System.out.println("iLg(10) = " + iLg(10));
    System.out.println("iLg(100) = " + iLg(100));
    System.out.println("iLg(1000) = " + iLg(1000));
    System.out.println("iLg(10000) = " + iLg(10000));
  }
  public static int iLg(int n)
  { int count=0;
    while (n > 1)
    { n /= 2;
      ++count;
    }
    return count;
  }
}
```

上述程序的输出为：

```
iLg(1) = 0
iLg(2) = 1
iLg(3) = 1
iLg(4) = 2
iLg(10) = 3
iLg(100) = 6
iLg(1000) = 9
iLg(10000) = 13
```

A.3　复杂性分类

在计算机科学中，算法按照复杂性函数进行分类。复杂性函数是用于描述算法相对于它所解决问题所需花费的时间。例如，冒泡排序算法的复杂性是 $O(n^2)$。如果使用排序算法中的数组增加为原来的 2 倍，则该算法的运行时间是原来的 4 倍：$(2n)^2 = 4n^2$。符号 $O(\)$代表数量级，$O(n^2)$表示 n^2 数量级。

“big Oh”表示法能够用极限术语来精确定义。若 f、g 是正整数集合上的非递减函数，则可用 $L(f, g)$表示极限：

标准分享网 www.bzfxw.com 免费下载

$$L(f,g)=\lim_{n\to\infty}\frac{f(n)}{g(n)}$$

极限的值可以等于 0、任何正整数或无穷大。例如，若 $f(n)=n^2$，且 $g(n)=n^{1/2}$，则 $L(f, g)=\infty$，因为 $n^2/\sqrt{n}=n^{3/2}\to\infty$。根据上述表示，我们可以有 5 种标准→的复杂性类型（如图 A.2）：

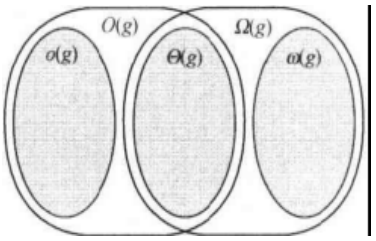


图 A.2 几个复杂性类型图的关系

$o(g)=\{f\in M \mid L(f,g)=0\}$

$O(g)=\{f\in M \mid 0\leqslant L(f,g)<\infty\}$

$\Theta(g)=\{f\in M \mid 0<L(f,g)<\infty\}$

$\Omega(g)=\{f\in M \mid 0<L(f,g)\leqslant\infty\}$

$\omega(g)=\{f\in M \mid L(f,g)=\infty\}$

其中，M 是所有自变量为正整数的正的单调递增函数。

定理 A.4

集合 $o(g)$、$\Theta(g)$和$\omega(g)$平分了空间 M，也就是说，M 中任何一个函数属于且仅属于这 3 个集合中的某一个。

定理 A.5

$O(g)=o(g)\cup\Theta(g)$

$\Theta(g)=O(g)\cap\Omega(g)$

$\Omega(g)=\Theta(g)\cup\omega(g)$

图 A.2 可以表示这 5 个集合间的关系。

注意，尽管这 5 种复杂性分类均是函数的集合，但我们通常还是这样书写：$f(n)=O(g(n))$，而不用$f\in O(g)$这样的精确表示。同理，用 $f(n)\neq O(g(n))$，而不写成$f\notin O(g)$。

例 A.4 复杂性分类

$n\lg n=O(n^2)$，但 $n\lg n\neq\Theta(n^2)$，因为$(n\lg n)/(n^2)=(\lg n)/n\to 0$。

$n\lg n=\Theta(n\lg n)$，因为$(n\lg n)/(n\log n)=(\log_2 n)/(\log_{10} n)=\log_{10}2$，是一个正数。

$n\lg n=\Omega(n^{1/2})$，但 $n\lg n\neq O(n^{1/2})$，因为$(n\lg n)/(n^{1/2})=n^{1/2}(\lg n)\to\infty$。

定理 A.6 对于任何底 $b>1$，$\log_b n=O(\lg n)$

证明：根据定理 A.2（f），$\log_b n=c\lg n$，$c=1/\lg b=\log_b 2$。

注意到，极限 $L(f, g)$可能不存在。在这种情况下，我们用 $L(f, g)$表示序列 $f(n)/g(n)$的各子序列极限的集合，这样，$L(f, g)$所需满足的不等式可以理解为集合中每个元素都满足的不等式。

例 A.5 振荡比率（oscillating ratio）

若 $f(n)=2^n$，$g(n)=4^{\lfloor n/2 \rfloor}$，则序列$\{f(n)/g(n)\}=\{1，2，1，2\cdots\cdots\}$没有极限值，因为 $L(f,g)$ 表示集合$\{1，2\}$。由于集合中的每个元素 x（也就是说对于 $x=1$ 和 $x=2$）均满足不等式 $o<x<\infty$，则我们认为 $f=\Theta(g)$（且 $g=\Theta(f)$）。

A.4 第一数学归纳法

第一数学归纳法，又称为弱归纳法，通常用于证明与自然数有关的公式。

定理 4.7（第一数学归纳法原理）

若$\{P(1),P(2),P(3),P(4),\ldots\}$是命题序列且满足下面两个性质，则所有命题均为真：

1. $P(1)$为真。
2. 任何命题均可以从它的前一个命题推导得出。

例 A.6 弱归纳法的运用

对于所有 $n\geqslant 1$，证明不等式 $2^n\leqslant(n+1)!$。

上述命题表示下列命题序列：

$P(1)$：$2^1 \leqslant 2!$

$P(2)$：$2^2 \leqslant 3!$

$P(3)$：$2^3 \leqslant 4!$

$P(4)$：$2^4 \leqslant 5!$

…

$P(1)$，$P(2)$，$P(3)$，$P(4)$ 4 个命题显然为真，这是因为：

$2^1=2 \leqslant 2=2!$

$2^2=4 \leqslant 6=3!$

$2^3=8 \leqslant 24=4!$

$2^4=16 \leqslant 120=5!$

符合条件的其他公式也可以用上述方法直接证明。但是我们无法这样证明，因为命题序列是无限的。

根据定理 A.7 的第 1 个条件，需要证明 $P(1)$为真，这已经在前面的陈述中给出了。定理的第 2 个条件需要从 $P(n-1)$归纳得出 $P(n)$。这样，我们可以假设对于任意 $n>1$，$P(n-1)$为真；也就是说对于任何 n，假设 $2^n\leqslant(n+1)!$，则下面需要做的工作是从假设中归纳得到 $P(n)$。首先考察 $P(n-1)$与 $P(n)$之间的关系：

$P(n-1)$：$2^{n-1}\leqslant n!$

$P(n)$：$2^n\leqslant(n+1)!$

标准分享网 www.bzfxw.com 免费下载

命题 $P(n)$又可以写成

$P(n) = (2) \cdot (2^{n-1}) \leqslant (n+1)\, n!$

现在，从 $P(n-1)$归纳得到 $P(n)$是显而易见的了。不等式左边乘以 2，右边乘以 $n+1$，主要 $n+1>2$ 即不等式左边乘以的因子小于右边乘以的因子，不等式就仍然成立。而对于任何 $n>1$，$n+1>2$。因此可得出：

$2^{n-1} \leqslant n! \Rightarrow 2^n \leqslant (n+1)!$

也就是说，$P(n-1) \Rightarrow P(n)$。

这满足定理 A.7 的第 2 个条件。因此我们得出结论，对于所有 $n \geqslant 1$，$P(n)$为真。

定理 A.7 的第一个条件称为归纳基础，第二个条件称为归纳步骤，第二个条件中的假设部分（即对于任何 n 假设命题 $P(n)$为真）称为归纳假设。

A.5 第二数学归纳法

第二数学归纳法称为“强归纳法”。它与第一数学归纳法非常类似，区别仅在于第二个条件中的归纳步骤上。

定理 A.8 第二数学归纳法原理

若{$P(1)$, $P(2)$, $P(3)$, $P(4)$, …}是满足下列两个条件的命题序列，则对于其他自然数，该命题序列均成立：

1. $P(1)$为真。
2. 任何命题均可以从它前面所有的命题推导得出。

归纳步骤（条件 2）的意思是 $P(n)$可以从前面所有命题假设{ $P(1), P(2), P(3), \ldots, P(n-1)$}推导得出。

例 A.7 算术基本定理的证明

定理 A.9 算法基本定理

每个正整数均可用质数的幂的乘积表示，且表示惟一：

$$p_1^{n_1} \cdot p_2^{n_2} \cdot p_3^{n_3} \cdot \ldots p_k^{n_k}$$

其中，p_1，p_2，p_3，…是质数：$p_1=2$，$p_2=3$，$p_3=5$ 等等，每个 n_j 是一个非负整数。

例如，对于正整数 23 115 456 有惟一的表示 $2^6 \cdot 3^4 \cdot 5^0 \cdot 7^3 \cdot 11^0 \cdot 13^1$。运用第二数学归纳法证明如下：

$P(1)$：“1 可以表示成质数的幂的乘积”

$P(2)$：“2 可以表示成质数的幂的乘积”

$P(3)$：“3 可以表示成质数的幂的乘积”

$P(4)$："4 可以表示成质数的幂的乘积"

…

上述 4 个命题为真，因为

$1=2^0$

$2=2^1$

$3=3^1$

$4=2^2$

上述命题的成立是归纳法的归纳基础，为了证明整个命题还需要完成归纳法的第二个步骤。因此我们假定对于任何 $n>1$，命题$\{P(1), P(2), P(3), \ldots, P(n-1)\}$为真。

若 n 有一个质数因子，用 p 表示，$m=n/p$，则 m 是一个非负整数，且 $m<n$，那么根据上述假设 $P(m)$为真，也就是说，m 可以用质数的幂的乘积惟一表示，而对于 n 来说，$n=p \cdot m$，因此在这种情况下 $P(n)$为真。

若 n 没有质数因子，则说明 n 本身是一个质数，因此它本身显然可以用质数的幂的乘积惟一表示：$n=n^1$，故在这种情况下，$P(n)$也为真。

现在，我们可以从命题$\{P(1), P(2), P(3), \ldots, P(n-1)\}$归纳出 $P(n)$。

A.6 等比级数

一个等比级数是一组数的和，其中每一项与前一项的比是一个常数。例如 10+30+90+270+810+2430+…是一个等比级数。因为每一项都是前一项的 3 倍。倍数 3 称为等比级数的公比。

定理 A.10 一个有限等比级数的和

$$a + ar + ar^2 + ar^3 + \ldots + ar^{n-1} = \frac{a(1-r^n)}{1-r}$$

这里，a 是等比级数的首项，r 是公比，n 是项的个数。

例 A.8 求一个有限等比级数的和

$$10+30+90+270+810+2430 = 10(1-3^6)/(1-3) = 10(1-729)/(-2) = 3\,640。$$

定理 A.11 无限等比级数的和

$$a + ar + ar^2 + ar^3 + \ldots = \frac{a}{1-r}$$

公式仅当 $-1<r<1$ 时成立。

例 A.9 求一个无限等比级数的和

$$6+3+3/2+3/4+3/8+\ldots = 6/(1-1/2) = 6/(1/2) = 12$$

标准分享网 www.bzfxw.com 免费下载

A.7 求和公式

定理 A.12 从 1 到 *n* 的 *n* 个正整数的和

$$1+2+3+\ldots+n=n(n+1)/2$$

注意，*n* 为和式中项的个数。

一种记忆该公式的简易方法是记忆图 A.3，图 A.3 中用点的形式表示两个三角形。一个三角形由白点组成，一个三角形由灰点组成。两个三角形有相同的点数。点数用 $S=1+2+3+\ldots+n$ 表示。两个三角形组成了一个 *n* 个点宽，*n*+1 个点高的矩形，因此点的个数是 *n*(*n*+1)，正好是 *S* 的 2 倍。

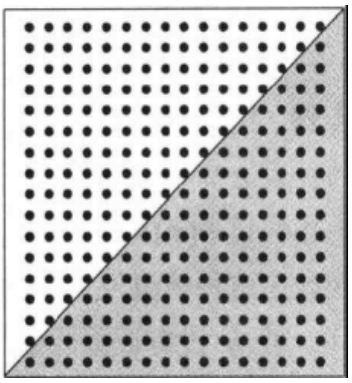


图 A.3 求和公式的记忆图

例 A.10 求一个数列的和

$$1+2+3+4+5+6+7+8 = 8(9) / 2 =36$$

定理 A.13 连续 n 个数的平方和

$$1^2+2^2+3^2+\ldots+n^2=\frac{n(n+1)(2n+1)}{6}$$

需要注意，尽管从等式的右边看是一个分数，但和式的结果肯定是一个整数。

例 A.11 求 *n* 个连续数的平方和

$$1+4+9+16+25+36+49 = 7\ (8)\ (15) / 6 = 140$$

A.8 调和数

调和数用下面的公式定义

$$H_n=\sum_{k=1}^{n}\frac{1}{k}=1+\frac{1}{2}+\frac{1}{3}+\frac{1}{4}+\frac{1}{5}+\ldots+\frac{1}{n}$$

前 6 个调和数见表 A.1。

表 A.1

n	*Hn*	*n*	*Hn*
1	1.000000	4	2.083333
2	1.500000	5	2.283333
3	1.833333	6	2.450000

调和数序列增长缓慢，然而，根据下面的定理和推论可知，该序列不可能无限增长。

定理 A.14 调和数序列是一个渐近对数

极限 $\lim_{n\to\infty}(H_n-\ln n)$ 是一个正数。

推论 A.1 调和数序列呈对数增长：$H_n=\Theta(\ln n)$。

下例是为定理 A.14 正确性证明提供了实验根据。

例 A.12 欧拉常数

为了验证极限 $\lim_{n\to\infty}(H_n-\ln n)$ 是一个正常数的猜测，我们运行下列程序并打印 $H_n-\ln n$ 的值，n 是 2 的指数，从 1 到 10 000 000。

```
public class ExA12
{ public static void main(String[] args)
  { double hn=0.0;            // harmonic numbers
    int pow2=1;               // powers of 2
    for (int n=1; n<1e7; n++)
    { hn += 1.0/n;            // the nth harmonic number
      if (n==pow2)            // print only for powers of 2
      { double ln=Math.log(n); // natural logarithm (base e) of n
        double difn = hn - ln; // approaches Euler's constant
        System.out.println(n+"\t"+hn+"\t"+ln+"\t"+difn);
        pow2 *= 2;
      }
    }
  }
}
```

上述程序的输出为：

```
1       1.0             0.0             0.8.68528194400547
2       1.5             0.6931471805599453      0.8068528194400547
4       2.083333333333333       1.3862943611198906      0.697038972213
8       2.7178571428571425      2.0794415416798357      0.638415601177
16      3.3807289932289937      2.772588722239781       0.608140270989
32      4.05849519543652        3.4657359027997265      0.592759292636
64      4.7438909037057675      4.1588830833596715      0.585007820346
128     5.433147092589174       4.852030263919617       0.581116828669
256     6.124344962817281       5.545177444479562       0.579167518337
512     6.81651653454972        6.238324625039508       0.578191909510
1024    7.509175672278132       6.931471805599453       0.577703866678
2048    8.202078771817716       7.6246189861593985      0.577459785658
4096    8.89510389696629        8.317766166719343       0.577337730246
```

标准分享网 www.bzfxw.com 免费下载

8192	9.588190046095265	9.010913347279288	0.577276698815
16384	10.281306710008463	9.704060527839234	0.577246182169
32768	10.974438632012168	10.39720770839918	0.577230923612
65536	11.667578183235785	11.090354888959125	0.577223294276
131072	12.360721549112862	11.78350206951907	0.577219479593
262144	13.053866822328144	12.476649250079015	0.577217572249
524288	13.747013049214582	13.16979643063896	0.577216618575
1048576	14.440159752936799	13.862943611198906	0.577216141737
2097152	15.133306695078193	14.556090791758852	0.577215903319
4194304	15.826453756428641	15.249237972318797	0.577215784109
8388608	16.51960087738358	15.942385152878742	0.577215724504

随着 n 的增大，$H_n-\ln n$ 的值趋近于常数 0.577 215 7。这个值就是所谓的欧拉常数，通常用希腊字母 γ 表示：

$$\gamma = \lim_{n\to\infty}(H_n - \ln n) = 0.577\,215\,7...$$

然而，γ 是否是个有理数仍然是个待探讨的问题。

A.9 Stirling 公式

阶乘函数出现在计算机科学的分析过程中。在这一领域，分析所关注的是包含 n!的表达式在 n 增大时所达到的数量级。但是这种表达式往往难以一目了然地得到其大小的度量，因为它们大都太大了。例如，$70! > 10^{100}$。Stirling 公式提供了一种估计阶乘大小的简便方法：

定理 A.15 Stirling 公式

$$1 < \frac{n!}{\sqrt{2n\pi}(\frac{e}{n})^n} < 1+\frac{1}{2n}$$

这里，e 和 π 分别是数学常量，$e = 2.718\,28$ 以及 $\pi = 3.141\,59$。

以下的程序表明，Stirling 公式的估计值是非常接近真实值的。

$$\sqrt{2n\pi}(\frac{n}{e})^n \le n! \le \sqrt{2n\pi}(\frac{n}{e})^{n+1/(12n)}$$

例 A.13 Stirling 公式的近似值

```
public class ExA13
{ public static void main(String[] args)
  { final double E=Math.E;      // e = 2.718 28..,
    final double PI=Math.PI;    // pi = 3.141 59..,
    double f=3628800.0;         // 10! = 3 628 800
    for (int n=10; n<200; n += 10)
```

```
    { double s = Math.sqrt(2*n*PI)*Math.pow(n/E,n);
      float r = (float)((f-s)/f);  // relative error
      System.out.println(n+"\t"+f+"\t"+s+"\t"+r);
      for (int i=1; i<=10; i++)
        f *= (n+i);
    }
  }
}
```

上述程序的输出为：

```
10    3628800.0      3598695.6187410373     0.008295961
20    2.43290200817664E18    2.4227868467611351E18   0.0041576526
30    2.6525285981219103E32   2.6451709592296516E32   0.0027738207
40    8.159152832478977E47    8.142172644946236E47    0.0020811213
50    3.0414093201713376E64   3.036344593938168E64    0.0016652563
60    8.320987112741392E81    8.3094383149767E81      0.001387912
70    1.197857166996989E100   1.1964320047337557E100  0.0011897598
80    7.156945704626378E118   7.14949447318118E118    0.0010411189
90    1.4857159644817607E138  1.4843409438918685E138  9.254936E-4
100   9.33262154439441E157    9.32484762526942E157    8.329834E-4
110   1.5882455415227421E178  1.587042784164154E178   7.572868E-4
120   6.689502913449124E198   6.68485904870404E198    6.942018E-4
130   6.466855489220472E219   6.462711405582573E219   6.4081897E-4
140   1.346201247571752E241   1.3454001771051692E241  5.9505994E-4
150   5.7133839564458505E262  5.7102107404794024E262  5.5540045E-4
160   4.714723635992059E284   4.7122686933249974E284  5.2069704E-4
170   7.257415615307994E306   7.253858934542908E306   4.900754E-4
```

注意，Stirling 公式的近似值可以精确到 1%以内，并且，随着 n 的增加，精确度相对提高。例如，当 n=100，$n!$=9.332 6(10^{178})，Stirling 公式的近似值为 9.324 8(10^{178})，相对误差为 0.08%。

推论 A.2　$n!=O(n^n)$。

证明：根据 Stirling 公式有

$$\frac{n!}{n^n} \le \sqrt{2n\pi}\left(\frac{n}{e}\right)^{1/(12n)}\left(\frac{1}{e}\right)^n$$

而表达式右边取极限为 0。

A.10　斐波那契数

斐波那契数是 1，1，2，3，5，8，13，21，34，55，…如图 A.4 所示。每个数是前两个数之和：

标准分享网 www.bzfxw.com 免费下载

$$F_n=\begin{cases}0, & 若\ n=0\\ 1, & 若\ n=1\\ F_{n-1}+F_{n-2}, & 若\ n>1\end{cases}$$

n	Fn	n	Fn
0	0	6	8
1	1	7	13
2	1	8	21
3	2	9	34
4	3	10	55
5	5		

图 A.4 斐波那契数

A.11 黄金分割

黄金分割（又称为黄金比）是一个数学常数：

$$\phi=\frac{1+\sqrt{5}}{2}\approx 1.618$$

该常数起源于一个古老的希腊问题：在一个线段 *AB* 内求一点 *C*，使得 *AB/AC* ≈ *AC/CB*，中点 *C* 称为黄金分割点，如图 A.5 所示。



图 A.5 黄金分割

定理 A.16 黄金分割

若点 C 是线段 AB 的黄金分割点，则 AB/AC=ϕ。

定理的证明（参见习题 A.11）显示，ϕ是一元二次方程 $x^2=x+1$ 的一个根，另一个根 $\psi=(1-\sqrt{5})/2=-0.618$ 称为共轭黄金分割。它与ϕ有下列特性。

定理 A.17 黄金分割特点

若 $\phi=(1+\sqrt{5})/2$，$\psi=(1-\sqrt{5})/2$，则

a. $\phi^2=\phi+1$

b. $\psi^2=\psi+1$

c. $1/\phi=\phi-1$

d. $1/\psi=\psi-1$

e. $\phi+\psi=1$

f. $\phi-\psi=\sqrt{5}$

黄金分割与斐波那契数有密切的联系。有如下结论：

定理 A.18 斐波那契数的精确公式

$$F_n = \frac{\varphi^n - \psi^n}{\sqrt{5}}$$

定理 A.18 中的公式是显而易见的。虽然ϕ、ψ、$\sqrt{5}$都是无理数，但是斐波那契数却是正整数。

推论 A.3 斐波那契数成近似指数：$F_n = \Theta(\phi^n)$。

A.12 欧几里德算法

欧几里德算法用于计算两个正整数的最大公约数。例如，494 与 130 的最大公约数 gcd 等于 26。因为 494 的公约数为{1，2，13，19，26，38，247，494}，130 的公约数为{1，2，5，10，13，26，130}。

算法 A.1 欧几里德算法

迭代形式的欧几里德算法如下：

```
public static long gcd(long a, long b)
{
  while (b>0)
  {
    long r = a % b;
    a = b;
    b = r;
    // 循环不变量：gcd(a, b)是常量
  }
  return a;
}
```

递归形式的欧几里德算法如下：

```
public static long gcd(long a, long b)
{
  if (b==0) return a;
  //循环不变量：gcd(a, b)是常量
  return (b, a%b);
}
```

例如，以下给出递归算法 gcd(494,130)的执行过程（如图 A.6 所示）：

下面的分析就是基于数论中的这一结论进行的。

定理 A.19 若 $a>b>0$，且 $r=a\%b>0$ 及 $d|b$，则 $d|a \Leftrightarrow d|r$

符合“|”表示 b 可被 d 整除，符合“%”表示 a 除以 b 的余数。

标准分享网 www.bzfxw.com 免费下载

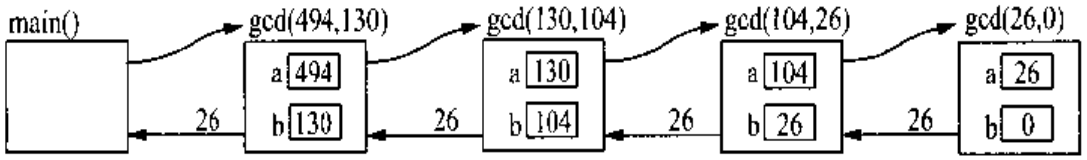


图 A.6 递归算法的执行过程

推论 A.4 若 $a > b > 0$，且 $r = a \% b > 0$，则 $(d \mid a) \wedge (d \mid b) \Leftrightarrow (d \mid b) \wedge (d \mid r)$

推论 A.5 若 $a > b > 0$，且 $r = a \% b > 0$，则 $\mathbf{gcd}(a, b) = \mathbf{gcd}(b, r)$。

例如，gcd(494, 130) = gcd(130, 104) = gcd(104, 26) = gcd(26, 0) = 26。

定理 A.20 欧几里德算法是正确的

算法中的不变量是 Q_k：$\gcd(a_k, b_k) = \gcd(a, b)$，$a_k$、$b_k$ 是 a、b 第 k 次循环的值。

定理 A.21 欧几里德算法的时间复杂性是对数量级的

欧几里德算法最多执行 $\log_\phi n$ 次除法，ϕ是黄金分割值，$n = \max\{a, b\}$。

A.13 Catalan 数

Catalan 数的递归定义如下：

$$C_n = \begin{cases} 1，若\ n=0 \\ C(0)\,C(n-1) + C(1)\,C(n-2) + C(2)\,C(n\ 3) + \ldots + C(n-1)\,C(0)\ ，若\ n>0 \end{cases}$$

前 10 个 Catalan 数见图 A.7，Catalan 数序列与斐波那契数序列相类似：它们都是递归定义的，且都呈指数增长。

n	C(n)	n	C(n)
0	1	5	42
1	1	6	132
2	2	7	429
3	5	8	1480
4	14	9	4862

图 A.7

例 A.14 Catalan 函数的实现

```
public static long cat(int n)
{
  if (n <2 ) return 1;
  long sum = cat(n-1);
  for (int i =1; i<n-1; i++)
    sum += cat(i) * cat(n-1-i);
  return sum;
```

}

Catalan 数是在运用第二数学归纳法解决下列问题时发现的。

例 A.15 计算一个多边形划分成三角形的方法的个数

一个多边形是由不相交线段围成的一个平面区域。一个凸多边形是指一个多边形的任意两点连线的线段都在多边形内部，因此，这种多边形的边界没有锯齿。图 A.8 是一些凸多边形的例子。

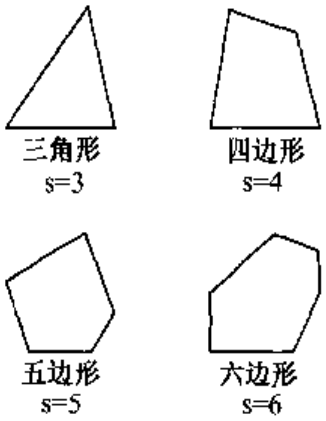


图 A.8

一个凸多边形的三角形划分就是把一个多边形划分为若干个三角形，所有三角形的角也是多边形的角。一个 S 条边的多边形的三角形划分是一个 $S-3$ 条非相交对角线的组合。图 A.9 是一个 6 边形的 4 种不同的三角形划分。

图 A.10 表明了一个四边形有两种三角形划分，一个五边形有 5 种三角形划分，一个六边形有 14 种三角形划分。从这些数据我们可以推测出下面定理的结论。

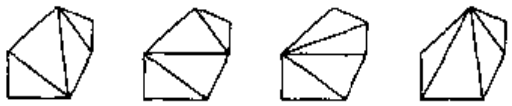


图 A.9 六边形的 4 种三角形划分

定理 A.22

一个有 $n+2$ 条边的多边形的三角形划分方法有第 n 个 Catalan 数 $C(n)$种。

证明：我们假定 $f(s)$表示一个 S 条的多边形的三角形划分的种数，则需要证明对于所有 $n \geq 1$，$f(n+2) = C(n)$。我们用强数学归纳法。

归纳法的基础是证明命题 $f(3) = C(1) = 1$，这个命题的证明是显而易见的：一个三角形只有一种三角形划分，就是它自己。

对于归纳步骤，我们假设对于 $n>1$，$k<n$，$f(k+2) = C(k)$成立。我们不知道 n 是多少，但是我们可以先假设 $n=8$，则我们需要证明 $f(10) = C(8)$，也就是说，一个 10 边形有 1 430 种不同的三角形划分方法。图 A.11 的最左边部分显示了一个 10 边形用一个三角形分成两部分：一个六边形和一个五边形。根据归纳假设，我们知道有 $f(6) = C(4) = 14$ 种不同的方法对六边形进行三角形划分，有 $f(5) = C(3) = 5$ 种不同的方法对五边形进行三角形划分，因此，就有 $C(4) \cdot C(3) = 14 \times 5 = 70$ 种不同的方法将 10 边形划分为三角形。同样的道理，第二个图表示有 $f(8) \cdot f(5) = C(6) \cdot C(3) = 132 \times 1 = 132$ 种不同的方法将 10 边形划分为三角形。最后一个图表明有 $f(9) = C(7) = 429$ 种不同的方法将 10 边形划分为三角形（阴影部分的三角形也包含在三角形划分中）。所有 $f(10)$的三角形划分都可以分成类似于上面的 3 个图的不同的类型来处理，每个类型的区别是阴影部分，而阴影部分的特殊三角形都有一条相同的边。所有

标准分享网 www.bzfxw.com 免费下载

f(10)的三角形划分可以分为 8 种类型，每种类型对应了 A—H 中的一个顶点。所以，10 边形的三角形划分总数为：

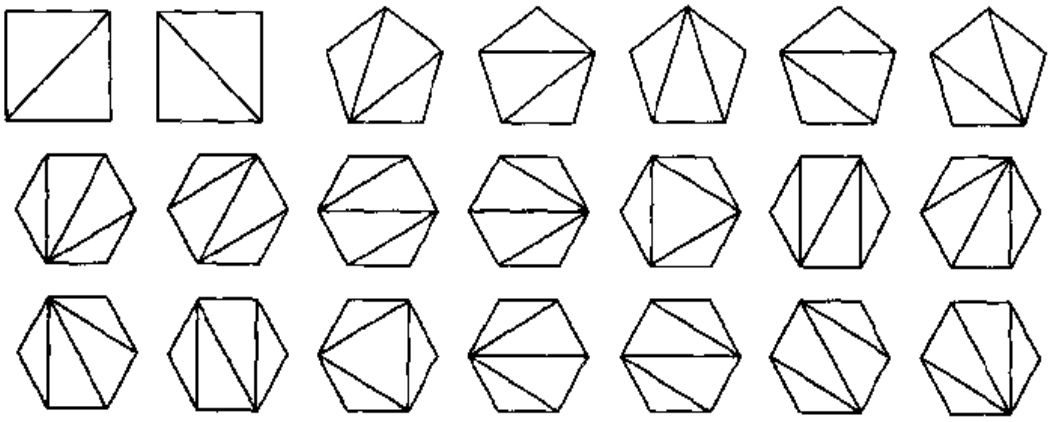


图 A.10 三角形划分

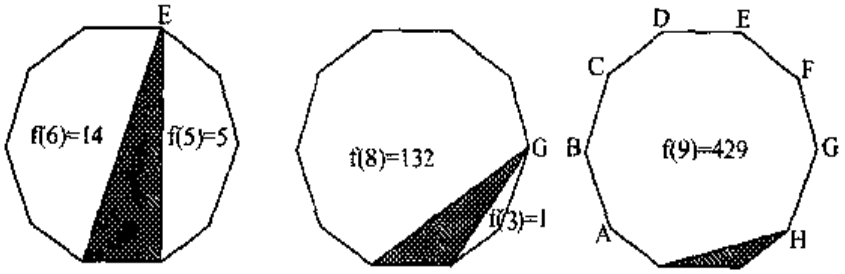


图 A.11

$f(10)=f(9)+f(3)\cdot f(8)+f(4)\cdot f(7)+f(5)\cdot f(6)+f(6)\cdot f(5)+f(7)\cdot f(4)+f(8)\cdot f(3)+f(9)$

由于 $f(s)=C(n-2)$，$C(0)=1$，有

$f(10)=C(7)+C(1)\cdot C(6)+C(2)\cdot C(5)+C(3)\cdot C(4)+C(4)\cdot C(3)+C(5)\cdot C(2)+C(6)\cdot C(1)+C(7)=C(8)$

我们可以用同样的论据证明对于所有 n，有 $f(n+2)=C(n)$。

定理 A.22 证明的方法称为“分治法”。这种方法被广泛应用于计算机科学中。这种方法的主要思想是，当你一旦分析了大小小于 n 的问题之后，你就可以分析大小为 n 的问题了：先将问题分成两个部分，再将以前的分析方法对每个部分进行分析。这样，分析整个问题就变成了分析整个问题与小问题之间的关系了。这种分析方法在折半查找（算法 2.2）、冒泡排序（算法 13.5）、快速排序（算法 13.6）中都得到了应用。

上述对 Catalan 数定义的公式是一个递归公式，除非已经知道了对于所有 $k<n$ 的 $C(k)$的值，否则不可能用这个公式来得到 $C(n)$的值。与此不同，下面给出的显示公式更加实用。

定理 A.23 Catalan 数的显示定理

$$C(n)=\frac{(2n)!}{n!(n+1)!}=\frac{(2n)(2n-1)(2n-2)...(n+2)}{(n)(n-1)(n-2)...(2)}$$

换句话说，上面的公式是在(2n) / (n)的基础上，分子、分母成对地添加因子，每次添加的因子都比前一项小 1，直到分母添加了因子 2 为止。

例 A.16　直接计算 Catalan 数

下面运用定理 A.23 验证例 A.13 的表中给出的 $C(9)$的值 4 862。

$$C(9)=\frac{(18)(17)(16)(15)(14)(13)(12)(11)}{(9)(8)(7)(6)(5)(4)(3)(2)}=(17)(13)(11)(2)=4\,862$$

由于 n=9，上式中分子、分母各有 8 项。

注意到，分母中的项在计算过程中都被约了。

定理 A.24　大小为 n 的二叉树的个数

大小为 n 的不同二叉树的个数是第 n 个 Catalan 数 $C(n)$。

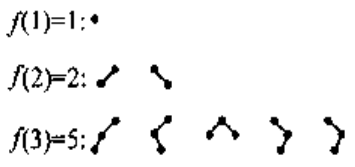


图 A.12　二叉数个数

证明：假设 $f(n)$表示大小为 n 的不同的二叉树的个数，则 $f(0) = 1 = C(0)$，因为显然大小为 0 的二叉树只有一种。图 A.12 表明 $f(1) = 1 = C(1)$，$f(2) = 2 = C(2)$，$f(3) = 5 = C(3)$，这些可以作为强数学归纳法的归纳基础（参见定理 A.8）。

假定对于所有 $n>0$，$k<n$，$f(k) = C(k)$成立。用 T 表示大小为 n 的二叉树。仿照定理 A.22 的证明，我们也采用“分治法”的策略。用 T_L 和 T_R 分别表示 T 的左子树和右子树，用 n_L 和 n_R 分别表示 T_L 和 T_R 的大小。因为 n_L 和 n_R 均小于 n，我们可以应用归纳假设 $f(n_L) = C(n_L)$和 $f(n_R) = C(n_R)$，而 $n=1+ n_L + n_R$。因为 T 中的每个点不是根节点就是在 T_L 中或 T_R 中，所以 $n_R = n - 1 - n_L$，且 n_L 可以是 $0\leqslant k\leqslant n-1$ 中任意一个数 k。所以大小为 n 的二叉树可分为 n 类，每一类对应 $n_L = k$（$0\leqslant k\leqslant n-1$）。

图 A.13 表明 n=8 时二叉树的分类：一类 n_L =3，另一类 n_L =5。在这种情况下，$n_L=k$，则 T_L 有 $C(n_L)$种不同的二叉树，T_R 有 $C(n_R)$种不同的二叉树，因此总共的二叉树种类有 $C(n_L) \cdot C(n_R) = C(k) \cdot C(n - 1 - k)$，二叉树总个数为 $f(n) = C(0)\,C(n-1) + C(1)\,C(n-2) + C(2)\,C(n-3) + \ldots + C(n-1)\,C(0) = C(n)$，这表明对于所有 n 有 $f(n) = C(n)$成立。


图 A.13

复　习　题

A.1　对于所有 x，若满足 f(f(x)) = f(x)，则称函数 f 为幂等函数，请解释下取整、上取整函数为幂等函数。

A.2　什么是对数？

A.3　弱归纳法与强归纳法的区别是什么？

A.4　如何确定什么时候使用强归纳法？

A.5　什么是欧拉常数？

标准分享网 www.bzfxw.com 免费下载

A.6 Stirling 公式特别适合于什么场合?

A.7 斐波那契数的递归定义和显示定义哪个更有用?

习 题

A.1 用图表示

$a.\ y = \lfloor x \rfloor$

$b.\ y = \lceil x \rceil$

$c.\ y = \lceil x \rceil - x$

$d.\ y = \lceil x \rceil - \lfloor x \rfloor$

A.2 证明定理 A.1。

A.3 证明定理 A.2。

A.4 证明定理 A.3。

A.5 判断真假:

$a.\ f = o(g) \Leftrightarrow g = \omega(f)$

$b.\ f = O(g) \Leftrightarrow g = \Omega(f)$

$c.\ f = \Theta(g) \Rightarrow g = \Theta(f)$

$d.\ f = O(g) \Rightarrow f = \Theta(g)$

$e.\ f = \Theta(g) \Rightarrow f = \Omega(g)$

$f.\ f = \Theta(h) \wedge g = \Theta(h) \Rightarrow f + g = \Theta(h)$

$g.\ f = \Theta(h) \wedge g = \Theta(h) \Rightarrow fg = \Theta(h)$

$h.\ n^2 = O(n \lg n)$

$i.\ n^2 = \Theta(n \lg n)$

$j.\ n^2 = \Omega(n \lg n)$

$k.\ \lg n = \omega(n)$

$l.\ \lg n = o(n)$

A.6 证明定理 A.4。

A.7 证明定理 A.5。

A.8 证明定理 A.10。

A.9 证明定理 A.11。

A.10 证明定理 A.12。

A.11 证明定理 A.16。

A.12 证明定理 A.17。

A.13 证明定理 A.18。

A.14 证明推论 A.3。

A.15 运行一个程序用于验证定理 A.18 公式，并将所得的值与斐波那契数的递归定义的值

进行比较。

A.16 证明定理 A.19。

A.17 证明推论 A.4。

A.18 证明推论 A.5。

A.19 证明定理 A.20。

A.20 证明定理 A.21。

复习题答案

A.1 下取整函数与上取整函数是幂等函数，这是因为函数返回整数，而根据定理 A.1e，整数的下取整函数和上取整函数是它本身。

A.2 对数也可以看成是指数，它是在给定的底上产生给定值的指数。

A.3 第一数学归纳法（弱归纳法）的归纳假设是假定对于某个 n 结论 $P(n)$成立。第二数学归纳法（强归纳法）的归纳假设是命题 $P(k)$对所有 k 小于等于 n 均成立。

A.4 命题 $P(n)$与 $P(n-1)$有直接联系的可使用弱归纳法，命题 $P(n)$依赖于 $P(k)$（$k<n-1$）则使用强归纳法。

A.5 欧拉常数是（$1+1/2+1/3+\ldots+1/n$）$-\ln n$ 的极限，它的近似值是 0.577 2。

A.6 Stirling 公式用于估计当 n 很大时，$n!$的值（例如当 $n>20$ 时）。

A.7 斐波那契数的递归定义（参见例 4.5）在 $n>20$ 时没有多大用途，因为递归调用次数呈指数增长，而显示公式（参见定理 A.18）在 $n<1\,475$ 时都非常有用。

习 题 解 答

A.1 函数 $y=\lfloor x\rfloor$ 、$y=\lceil x\rceil$、$y=\lceil x\rceil-x$ 及 $y=\lceil x\rceil-\lfloor x\rfloor$的图示如图 A.14：

A.2 定理 A.1 的证明如下：

a. 关系 $\lfloor x\rfloor = \max\{ m\in\mathbf{Z} \mid m\leqslant x \}$且 $\lceil x\rceil = \min\{ n\in\mathbf{Z} \mid n\geqslant x \}$实际上是$\lfloor x\rfloor$ 以及$\lceil x\rceil$的定义的另外一种陈述。

b. 令 $m=\lfloor x\rfloor$且 $n=\lceil x\rceil$，则根据定义，$m\leqslant x<m+1$，且 $n-1<x\leqslant n$，则 $x-1<m$ 且 $n<x+1$，因此，$x-1<m\leqslant x\leqslant n<x+1$。

c. 不等式 $x-1<\lfloor x\rfloor\leqslant x\leqslant\lceil x\rceil<x+1$ 实际上是对 b 中内容的总结。

d. 令 $n\in\mathbf{Z}$ 使得 $n\leqslant x\leqslant n+1$，令 $A=\{m\in\mathbf{Z} \mid m\leqslant x\}$，则 $n\in A$ 且$\lfloor x\rfloor=\max A$，因此 $n\leqslant\lfloor x\rfloor$。如果 $n<\lfloor x\rfloor$，则 $n+1\leqslant\lfloor x\rfloor$（由于 n 和$\lfloor x\rfloor$都是整数），但是根据假设，$n+1\geqslant\lfloor x\rfloor$，则有 $n=\lfloor x\rfloor$。定理的第二部分的证明类似。

e. 假设 $x\in\mathbf{Z}$（也就是说 x 是一个整数），令 n 等于 d 中的 x，则有 $x\leqslant x<x+1$ 则 $x=\lfloor x\rfloor$，且有 $x-1<x\leqslant x$ 则 $x=\lceil x\rceil$。

f. 假设 $x\notin\mathbf{Z}$（也就是说 x 不是一个整数），令 $u=x-\lfloor x\rfloor$，$v=\lceil x\rceil-x$，那么，根据 c 的

标准分享网 www.bzfxw.com 免费下载

结论有 $u \geq 0$ 且 $v \geq 0$，同样还是根据 c 的结论，有 $x-1<\lfloor x \rfloor = x-u$ 且 $v+x=\lceil x \rceil < x+1$，因此，$u<1$ 且 $v<1$。因此有 $0 \leq u<1$ 且 $0 \leq v<1$，但是由于 u 和 v 都不可能是整数，因为如果它们中任意一个是整数，则 x 也必然是整数，这是因为有 $x=\lfloor x \rfloor + u = \lceil x \rceil - v$ 成立。因此，$0<u<1$ 且 $0<v<1$。因此，$x=\lfloor x \rfloor + u > \lfloor x \rfloor$，$x=\lceil x \rceil - v < \lceil x \rceil$。

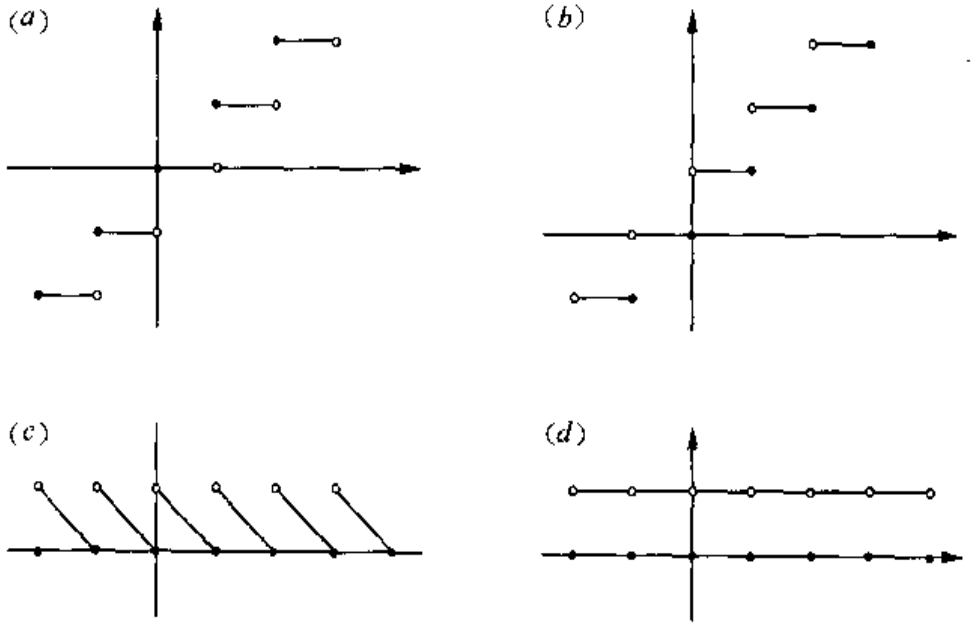


图 A.14 习题 A.1 图

g. 令 $n=-\lfloor -x \rfloor$，则 $-n=\lfloor (-x) \rfloor$，根据 c 的结论，$(-x)-1<(-n) \leq (-x)$，因此，$x \leq n<x+1$，则 $x \leq n$ 且 $n-1<x$，因此，$n-1<x \leq n$。根据 d 的结论，$n=\lceil x \rceil$。因此，$-\lfloor -x \rfloor = \lceil x \rceil$，因此，$\lfloor -x \rfloor = -\lceil x \rceil$。

h. 令 $n=\lfloor x+1 \rfloor$，则根据 c 的结论，$(x+1)-1<n \leq (x+1)$，因此，$x-1<n-1 \leq x$ 且 $x=(x+1)-1<n$。近而有 $n-1 \leq x<n$，也就是说，$(n-1) \leq x<(n-1)+1$，根据 d 的结论，有 $(n-1)=\lfloor -x \rfloor$，$\lfloor x+1 \rfloor = n = \lfloor x \rfloor + 1$。第二个等式的证明类似。

A.3 定理 A.2 的证明如下：

a. 令 $x=b^y$，则根据定义，$\log_b(b^y)=\log_b(x)=y$。

b. 令 $y=\log_b x$，则根据定义，$b^{\log_b x}=b^y=x$。

c. 令 $y=\log_b u$，$z=\log_b v$，则根据定义，$u=b^y$，$v=b^z$，因此 $uv=(b^y)(b^z)=b^{y+z}$，因而，$\log_b(uv)=y+z=\log_b u+\log_b v$。

d. 根据 c 的结论，$\log_b v+\log_b u/v=\log_b(v\,u/v)=\log_b u$，因此有 $\log_b u/v=\log_b u-\log_b v$。

e. 令 $y=\log_b u$，则根据定义，$u=b^y$，$(u)^v=(b^y)^v=b^{vy}$，根据定义 $\log_b(u^v)=vy=v\log_b u$。

f. 令 $y=\log_b x$，则根据定义，$x=b^y$，$\log_c x=\log_c(b^y)=y\log_c b$，根据 e 的结论有 $\log_b x=y=(\log_c x)/(\log_c b)$。

A.4 定理 A.3 的证明如下：

a. 已知 $2^p<n<2^{p+1}$，其中 n、$p \in \mathbf{Z}$，令 $x=\lg n$，对该表达式各部分同时取以 2 为底的

对数可得到 $p < x < p+1$，再根据定理 A.1*d*，$p=\lfloor x \rfloor = \lfloor \lg n \rfloor$。类似地，$(p+1)-1<x<(p+1)$，因而也有 $p+1=\lceil \lg n \rceil$。

b. 令 $x = \lg(n+1)$且 $y = \lg n$。根据 A.1*c* 的结论，$y < \lfloor y \rfloor + 1$，因此 $n = 2^y < 2^{\lfloor y \rfloor + 1}$。但是现在不等式 $n < 2^{\lfloor y \rfloor + 1}$的两边都是整数，因此有 $n+1 \leqslant 2^{\lfloor y \rfloor + 1}$，进而有 $x = \lg(n+1) \leqslant \lfloor y \rfloor + 1$。但同时有 $x = \lg(n+1) \geqslant \lg n = y \geqslant \lfloor y \rfloor$，故 $m-1 < x \leqslant m$，其中 $m = \lfloor y \rfloor + 1 = \lfloor \lg n \rfloor + 1$。因此，根据 A.1*d* 的结论，$\lceil x \rceil = m$，也就是说，$\lceil \lg(n+1) \rceil = \lfloor \lg n \rfloor + 1$。

A.5 判断真假：

a. 真

b. 真

c. 真

d. 假

e. 真

f. 真

g. 假

h. 假

i. 假

j. 真

k. 假

l. 真

A.6 定理 A.4 的证明如下：

对于 M 中的给定函数 f 和 g，表达式 L(f, g)必然是 0、正数或负数 3 种情况之一。这 3 种互不相交的情况决定了 f 是分别属于 o(g)、Θ(g)还是ω(g)。

A.7 定理 A.5 的证明如下：

$$
\begin{aligned}
&O(g) = \{f \in M \mid 0 \leqslant L(f, g) < \infty\} \\
&= \{f \in M \mid L(f, g) = 0 \text{ 或 } 0 < L(f, g) < \infty\} \\
&= \{f \in M \mid L(f, g) = 0\} \cup \{f \in M \mid 0 < L(f, g) < \infty\} \\
&= o(g) \cup \Theta(g)
\end{aligned}
$$

$$
\begin{aligned}
&\Theta(g) = \{f \in M \mid 0 < L(f, g) < \infty\} \\
&= \{f \in M \mid L(f, g) > 0 \text{ 且 } L(f, g) < \infty\} \\
&= \{f \in M \mid L(f, g) > 0\} \cap \{f \in M \mid L(f, g) < \infty\} \\
&= O(g) \cap \Omega(g)
\end{aligned}
$$

$$
\begin{aligned}
&\Omega(g) = \{f \in M \mid 0 < L(f, g) \leqslant \infty\} \\
&= \{f \in M \mid 0 < L(f, g) < \infty \text{ 且 } L(f, g) = \infty\} \\
&= \{f \in M \mid 0 < L(f, g) < \infty\} \cap \{f \in M \mid L(f, g) = \infty\}
\end{aligned}
$$

标准分享网 www.bzfxw.com 免费下载

$=\Theta(g)\cup\omega(g)$

A.8 定理 A.10 的证明如下：

令 $S=a+ar+ar^2+ar^3+\ldots+ar^{n-1}$，则 $rS=ar+ar^2+ar^3+ar^4+\ldots+ar^n$，因此 $S-rS=a-ar^n$，$(1-r)S=a(1-r^n)$，因此有 $S=a(1-r^n)/(1-r)$。

A.9 定理 A.11 的证明如下：

如果 $-1<r<1$，则当 n 无限增大时，r^n 将趋近于 0。在定理 A.10 中令 $r^n=0$ 即可得到结论。

A.10 定理 A.12 的证明如下：

令 $S=1+2+3+\ldots+n$，$S=n+\ldots+3+2+1$ 自然成立。将这两个等式相加，将等式右边的 $2n$ 个数两两相加，将得到 $n(n+1)$，等式左边为 $2S$，因此 $S=n(n+1)/2$。

A.11 定理 A.16 的证明如下：

如果 $r=AC/CB$，则 $r=AC/CB=AB/AC=(AC+CB)/AC=1+CB/AC=1+1/r$，因此 $r^2=r+1$。解这个方程得到 $r=(1\pm\sqrt{5})/2$，两个根的数值分别为：

$r_1=(1+\sqrt{5})/2=\phi=1.618\,033\,988\,749\,894\,848\,204\,586\,834\,365\,6\ldots$

$r_2=(1-\sqrt{5})/2=\psi=-0.618\,033\,988\,749\,894\,848\,204\,586\,834\,365\,6\ldots$

因为黄金分割值 $\phi>0$，因此它是正确的答案。

A.12 定理 A.17 的证明如下：

a. $\phi^2=\phi+1$，因为 ϕ 是方程 $x^2=x+1$ 的一个解。

b. $\psi^2=\psi+1$，因为 ψ 是方程 $x^2=x+1$ 的一个解。

c. $1/\phi=\phi-1$，因为 $1=\phi^2-\phi$。

d. $1/\psi=\psi-1$，因为 $1=\psi^2-\psi$。

e. $\phi+\psi=(1+\sqrt{5})/2+(1-\sqrt{5})/2=1$。

f. $\phi-\psi=(1+\sqrt{5})/2-(1-\sqrt{5})/2=\sqrt{5}$。

g. A.13 定理 A.18 的证明如下：

$$F_0=\frac{\phi^0-\psi^0}{\sqrt{5}}=\frac{1-1}{\sqrt{5}}=0$$

$$F_1=\frac{\phi^1-\psi^1}{\sqrt{5}}=\frac{\phi-\psi}{\sqrt{5}}=\frac{\sqrt{5}}{\sqrt{5}}=1$$

$$F_{n+1}=\frac{\phi^{n+1}-\psi^{n+1}}{\sqrt{5}}=\frac{(\phi^n+\phi^{n-1})-(\psi^n+\psi^{n-1})}{\sqrt{5}}=\frac{\phi^n-\psi^n}{\sqrt{5}}+\frac{\phi^{n-1}-\psi^{n-1}}{\sqrt{5}}=F_n+F_{n-1}$$

A.14 推论 A.3 的证明如下：

由于 $-1<\psi<0$，ψ 的高次幂将被忽略，因此，$F_n=k\phi^n$，其中 $k=1/\sqrt{5}$。

A.15 程序如下：

```
public class PrA15
{ public static void main(String[] args)
```

```
{ final double SQRT5 = Math.sqrt(5.0);
  final double PHI = (1 + SQRT5)/2;
  final double PSI = (1 - SQRT5)/2;
  long f0, f1=0, f2=1;
  for (int n=2; n<32; n++)
  { f0 = f1;
    f1 = f2;
    f2 = f1 + f0;
    float fn = (float)((Math.pow(PHI,n)-Math.pow(PSI,n))/SQRT5);
    System.out.println(f2+"\t"+fn);
  }
 }
}
```

A.16 定理 A.19 的证明如下：

假设 $a>b>0$ 且 $r=a\%b>0$ 且 $d \mid b$，则对于某个 $q\in\mathbf{N}$ 有 $a=qb+r$，且对于某个 $k\in\mathbf{N}$ 有 $b=kd$，因此 $a=qb+r=qkd+r$。则对于某个 $m\in\mathbf{N}$ 有 $d \mid a \Rightarrow a=md \Rightarrow md=a=qkd+r \Rightarrow r=md-qkd=(m-qk)d \Rightarrow d \mid r$。类似地有，对于某个 $n\in\mathbf{N}$ 有 $d \mid r \Rightarrow r=nd \Rightarrow a=qkd+r=qkd+nd=(qk+n)d \Rightarrow d \mid a$。

A.17 推论 A.4 的证明如下：

这是逻辑表达式 $(P\wedge Q)\rightarrow(R\leftrightarrow S)\Rightarrow P\rightarrow(Q\wedge R\leftrightarrow Q\wedge S)$ 的推论，其 $P=$“$a>b>0$ 且 $r=a\%b>0$”，$Q=$“$d \mid b$”，$R=$“$d \mid a$”，$S=$“$d \mid r$”。

A.18 推论 A.5 的证明如下：

令 $A=\{d\in\mathbf{N}: d \mid a$ 且 $d \mid b\}$ 且令 $B=\{d\in\mathbf{N}: d \mid b$ 且 $d \mid r\}$，根据推论 A.4，有 $A=B$。所以，$\max A=\max B$，但是根据定义 $\max A=\gcd(a, b)$ 且 $\max B=\gcd(b, r)$。

A.19 定理 A.20 的证明如下：

推论 A.4 保证了循环的不变量总为真。循环在 $b=0$ 时停止，并返回 a，即 $\gcd(a, b)$。因此，返回的值也是最初的 (a, b) 的 gcd 值。

A.20 定理 A.21 的证明如下：

在第 k 次循环中，$a_k=q_kb_k+r_k$，其中，$q_k=a_k/b_k\geq 1$ 且 $r_k=a_k\%b_k$，因此，$a_k\geq b_k+r_k$，但是 $b_k=a_{k+1}$ 且 $r_k=b_{k+1}=a_{k+2}$，所以 $a_k\geq a_{k+1}+a_{k+2}$。现在考虑逆序列 $\{x_i\}$，其中，$x_0=a_m$，$x_i=a_{m-i}$，对于所有的 i 有 $x_0\geq 1$，$x_1\geq 1$，$x_i\geq x_{i-1}+x_{i-2}$，因此有 $x_2\geq x_1+x_0\geq 1+1=2$，$x_3\geq x_2+x_1\geq 2+1=3$，等等。因此 $x_i\geq F_{i+1}$（斐波那契数）。所以，$a=x_m\geq F_{m+1}\geq \varphi^m$，进而得到 $\log_\varphi a\geq m$。

标准分享网 www.bzfxw.com 免费下载

附录B　从C++到Java

本附录介绍Java与C++中的有关概念及其对应关系。这对于正在学习Java的C++程序员将很有帮助。

Java与C++的主要区别在于：

- 在Java中，所有执行语句都要被封装在类中。
- 在Java中，使用Object元素来替代C++中的template类。
- 在Java中，不允许有外部函数和变量。
- 在Java中，使用引用的概念，而不是指针的概念。
- 在Java中，所有参数传递都是传值方式。
- C++中的下列定义

```
string* s = new string;
```

等价于Java中的如下定义

```
String s = new String();
```

- Java使用自动垃圾回收机制，而不用delete运算符。

下面给出C++同Java的一些对应关系：

bool ····· boolean
char ····· N/A
wchar_t ····· char
short ····· short
int ····· int
long ····· long
unsigned char ····· byte
unsigned short ····· N/A
unsigned int ····· N/A
unsigned long ····· N/A
float ····· float
double ····· double
enum ····· N/A
string ····· String
const ····· final

goto ····· N/A
指针 ····· N/A
引用 ····· N/A
参数传值 ····· 参数传递
参数传址 ····· N/A
inline ····· N/A
register ····· N/A
namespace ····· package
流I/O ····· N/A
printf() ····· System.out.println()
scanf() ····· N/A
class ····· class
struct ····· class
数据成员 ····· 域，实例变量
成员函数 ····· 方法
public ····· public
protected ····· protected
private ····· private
static ····· static
this ····· this
new ····· new
delete ····· N/A
template ····· N/A
operator ····· N/A
friend ····· N/A
sizeof ····· N/A
typeof ····· N/A
typeid ····· getClass()
virtual ····· abstract
virtual class ····· abstract class
virtual 函数 ····· abstract 方法
纯virtual类 ····· interface
头文件 ····· 包
#include ····· import
vector ····· Vector
stack ····· Stack

标准分享网 www.bzfxw.com 免费下载

queue ····· N/A

deque ····· N/A

priority_queue ····· N/A

list ····· LinkedList

map ····· HashMap

set ····· HashSet

附录 C　Java 开发环境

开发环境是支持用户开发新的计算机程序的一组计算机程序。任何开发工具中的两个关键部件是文本编辑器和编译器。商业化的集成开发环境（IDE）还包括调试器，编程语言的文档以及其他工具。

C.1　Windows 的命令窗口

Java 的最简单的开发环境可以免费获得。可以使用绑定在计算机操作系统中的文本编辑器，并从 Sun 公司的 Web 网站 `http://java.sun.com` 下载最新版本的 Java。

如果使用的是 Microsoft Windows，那么可以使用记事本软件或写字板软件。首先创建一个名为 `Hello.java` 的文本文件，并将它存入 A 盘。然后从“开始”按钮中的“程序→附件”菜单中打开 Windows 命令窗口，在命令窗口中利用 javac 编译 `Hello.java`，利用 java 执行编译后得到的字节码。需要注意的是，在编译和执行之前，要根据需要设置环境变量 `PATH`，目的是让操作系统能够找到 `javac` 和 `java` 这两个命令。

如果要了解更多的细节，请参见参考文献[Hubbard]条目。

C.2　Visual Cafe

Visual Cafe 最初是 Symantec 公司的产品，但是现在已经是 WebGain 公司的产品了。其 4.0 版本是 2000 年 8 月发布的。

在 Visual Cafe 中，要创建一个标准的 Java 应用程序，要经过以下步骤：

1. 从 File 菜单下选取 New Project...（或者使用快捷键 Ctrl+Shift+N），然后在弹出的对话框（如图 C.1 所示）中选择 Win32 Console Application。

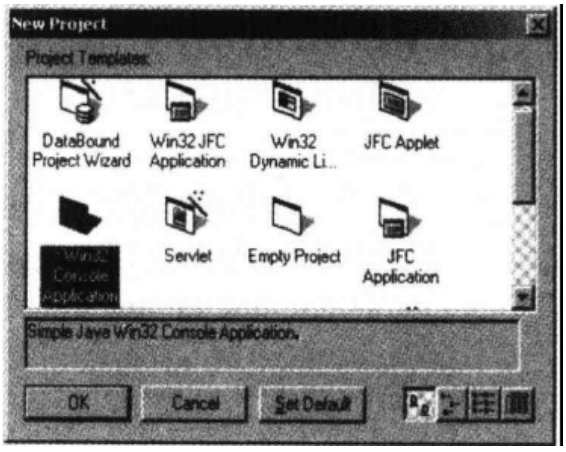


图 C.1　创建新的工程

2. 这将创建一个新的工程，并定义一个名为 `SimplCon` 的类：

```
/*
   A basic Java class stub for a Win32 Console Application.
*/
```

标准分享网 www.bzfxw.com 免费下载

```
public class SimplCon {
    public SimplCon() {
    }
    static public void main(String args[]) {
      System.out.println("Hello World!");
   }
}
```

这是一个典型的“Hello World!”应用程序。

3. 为了运行这个程序，从 Project 菜单中选择 Execute（或者使用快捷键 Ctrl+F5），这个命令将编译程序并执行它，并在一个标准的 DOS 窗口中输出结果，如图 C.2 所示。

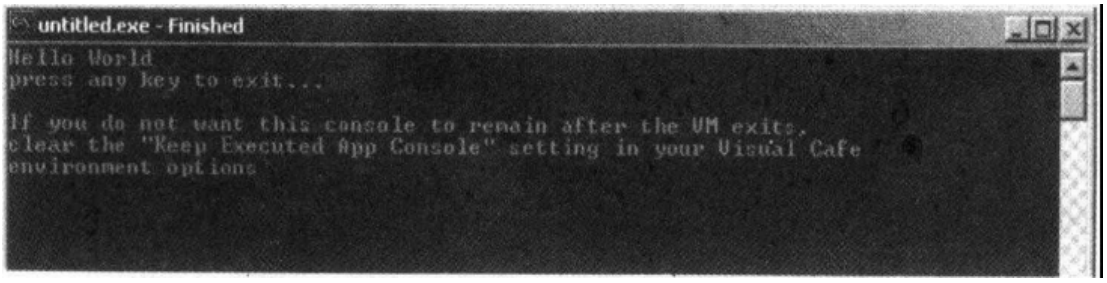


图 C.2 程序的运行结果

4. 按“回车键”关闭 DOS 窗口。
5. 如果要保存当前的工程，用鼠标点击工程窗口（该窗口标题为“Project—Untitled”），然后从 File 菜单中选择 Save As...。在本例中，我们将工程命名为 Project1.vep 保存在 A 盘上。系统将自动地将 SimplCon.java 保存在相同目录下。
6. 现在我们来为主类改个名字。这不但需要更改源代码窗口中所有出现 SimplCon 的字符串，还要利用 File 菜单中的 Save As...来更改这一文件的名字。在本例中，我们将主类改名为 Main。注意到，这时工程中存在两个类，即 SimplCon 和 Main，它们都有自己的 main()方法。双击 SimplCon 类的文件图标，在编辑窗口中重新打开它。
7. 对于一个工程来说，只能有一个包含main()方法的类，也就是说，这个惟一的main()方法将是程序执行的起点。为了区分当前工程中的两个 main()方法，哪一个是执行的起点，首先将 Main 类中的 println 方法中的字符串改为“Goodbye, World!”。然后，按 Ctrl+F5 来执行它（或者从 Project 菜单中选择 Execute）。程序的输出应该还是“Hello, World!”，同前一次一样，这表明程序的执行起点没有改变。
8. 为了改变程序的执行起点，从 Project 菜单中选择 Options...，将弹出 Project Options 窗口。将 Main Class 一项由 SimplCon 改为 Main，然后按 OK 按钮返回工程。此时再执行程序，就会得到“Goodbye, World!”的输出信息了。
9. 现在关闭 SimplCon.java 窗口，然后用鼠标右键点击它在工程窗口中的图标，从

弹出的菜单中选择 Cut（或者从 Edit 菜单中选择 Cut）将该文件从工程中删除。现在，工程中就只剩下新生成的 Main 类了。

10. 现在修改 Main 类，加入另外一条 println 语句：

```
static public void main(String args[]) {
    System.out.println(args[0] + " **** " + args[1]);

    System.out.println("Goodbye, World!");
}
```

然后，再次打开 Project Options 窗口（或者选择从 Project 菜单中选择 Options...），将字符串“Hello World!!!!”填入 Program Arguments 域，如图 C.3 所示，然后按 OK 按钮。再次运行程序，输出结果将为：

```
Hello **** World!!!!
Goodbye, World!
```

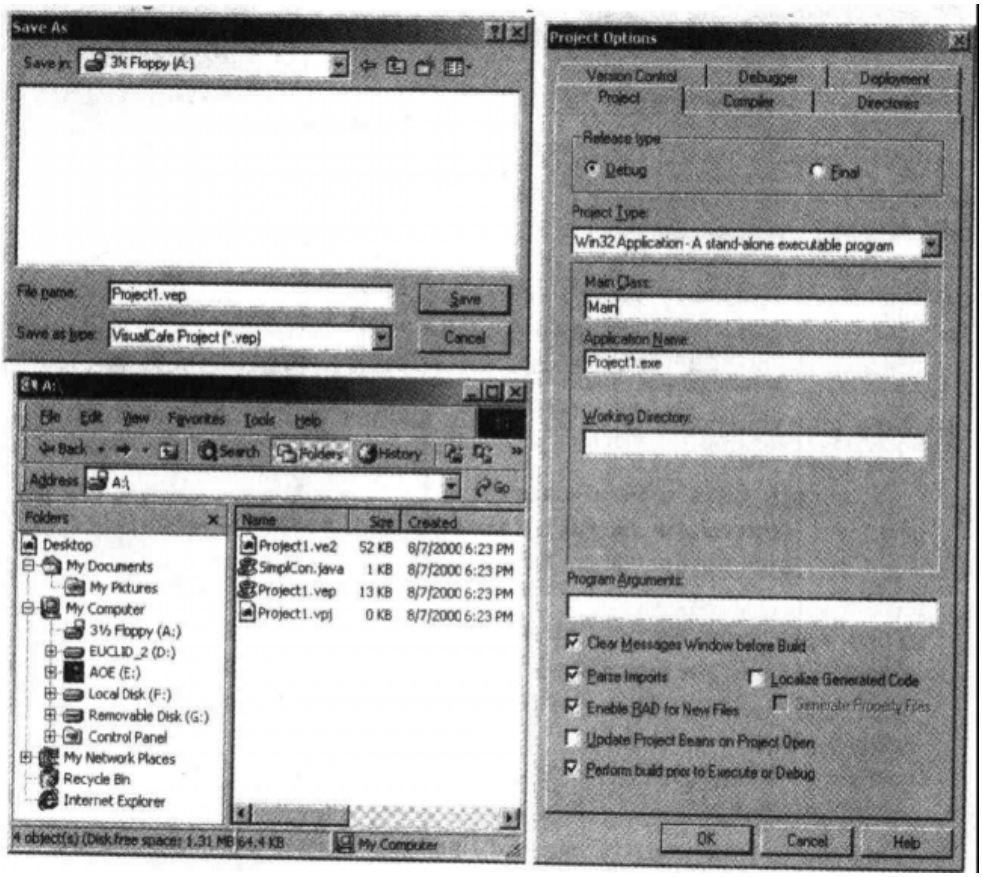


图 C.3 Project options 窗口

这向我们展示如何使用命令行参数，在 Program Arguments 域中的每一个字符串都成为 main()方法的参数数组 args[]的一个数组元素。

11. 为了向工程中加入一个新类，从 Insert 菜单中选择 Class...。将该类命名为 Widget 并在 Package 域中填入 DSwJ.util，如图 C.4 所示。然后按 Finish 按钮，这

标准分享网 www.bzfxw.com 免费下载

时将弹出一个对话框询问是否创建新目录，按 Yes 表示同意创建。然后查看一下工程目录，看一看是否有新创建的目录。Java 的包变成了对应的目录路径，包的每一部分都是一个嵌套的文件夹。因此，包 DSwJ.util 将创建一个名为 util 的文件夹，它嵌套在文件夹 DSwJ 中，而新创建的 Widget.java 被放置在内层的 util 文件夹内。

12. 双击工程窗口中的 Widget 图标，将打开该类对应的源代码文件。它包含了最少的类的声明：

```
package DSwJ.util;
public class Widget
{
}
```

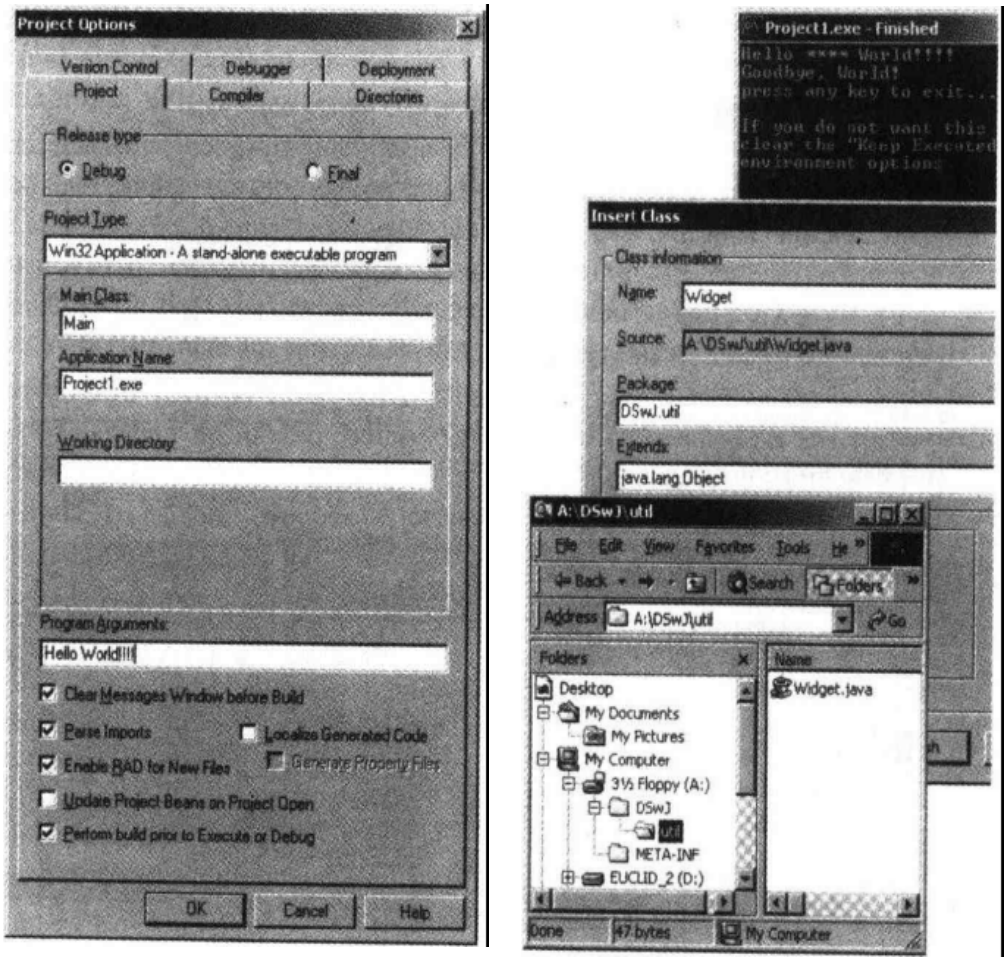


图 C.4 活动窗体

13. 单击工程窗口底部的 Package 标签切换到 Package 视图。在视图的顶部是处于 Default Package 包下的 Main.java 文件，然后是处于 DSwJ.util 包下的 Widget.java 文件。在这些内容以下，就是一长串 Java 标准类库的包名。向下

滚动直到找到 java.util 包，点击并展开其中定义的所有类。双击 Stack.java 将打开该文件的源代码。

14. 从 View 菜单中选择 Class Browser（或者使用快捷键 Ctrl+Shift+C），将打开 Visual Cafe Class Browser 窗口。它包含 3 个部分：Classes 列表，Member 窗格区，以及编辑窗格区。用鼠标右键打开 Classes 列表，然后选择 Collapse All 来清除所有文件夹列表。然后像第 13 步一样，找到 java.util 包，展开包并双击 Stack 项。这时，Member 窗格区将列出 java.util.Stack 类的所有成员。单击 push() 方法将使得其源代码显示于编辑窗格区。

Visual Cafe 具备其他所有 IDE 通常应该具备的特点：

- 带彩色标注功能的编辑器。
- 基于按钮控制的调试器。
- 完善的联机帮助。

它还具备一些独特的特点，例如：

- 含义明确丰富的编译器消息。
- 帮助程序员进行类声明的 class wizard。
- 帮助程序员正确、快速编码的 code helper。
- 语法检查器。
- 层次编辑器。
- 支持用户自定义宏。

Visual Cafe 的联机帮助系统非常好，如图 C.5 所示。Help 菜单中的 Java API reference...命令提供目前所有包、类、域、方法等的列表说明。其中包括 Java 标准的库和 WebGain 的扩展库。Class Hierarchy 窗口展示了自 Object 类而下的各个类的层次结构。Index of all Classes, Fields, and Methods 包含了 20 000 多个表项。Package Index 中罗列了来自 Java、Sun 和 Symantec 的所有 95 个包。通过 API Reference 窗口顶部的标签，可以自由地在这些索引间来回切换。所有索引都使用超文本形式，方便程序员阅读。遗憾的是，API Reference 并不是目前的最新版本。2000 年 8 月的发行版本不包含 Java 1.2 中的一些类，比如 java.util.Arrays 等。

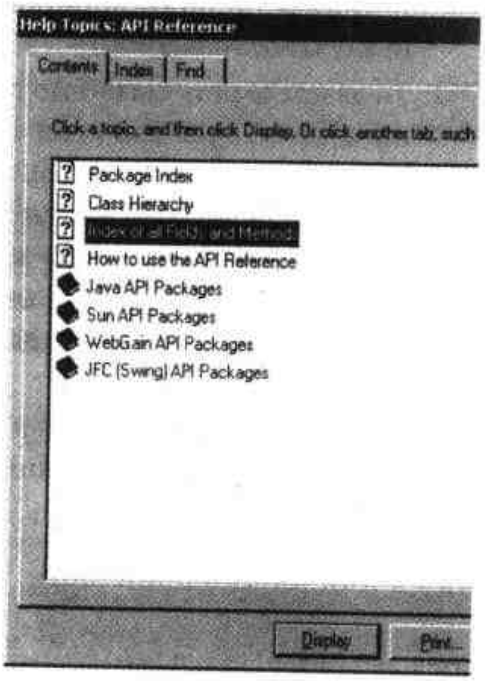


图 C.5 Visual Cafe 的帮助信息

标准分享网 www.bzfxw.com 免费下载

附录D 参考文献

出版商：

Addison-Wesley Publishing Company, Reading, MA, http://www.awl.com/cs/
John Wiley & Sons, New York, NY, http://www.wiley.com/college/eng/comsci/
Jones and Bartlett Publishers, Sudbury, MA, http://www.jbpub.com/cs/
Macmillan, New York, NY, http://www.macmillan.com
McGraw-Hill, New York, NY, http://www.mhhe.com
O'Reilly & Associations, Inc., Sebastopol, CA, http://www.oreilly.com
Oxford University Press, New York, NY, http://oup-usa.com
Perseus Books, Reading MA, http://www.perseusbooksgroup.com
Prentice-Hall, Englewood Cliffs, NJ, http://www.prenhall.com
PWS Publishing Company, Boston, MA, http://www.brookscole.com
Que Education & Training, Indianapolis, IN, http://ww.mcp.com/resources/
SYBEX, Alameda, CA, http://www.sybex.com/
WCB/McGraw-Hill, New York, NY, http://www.mcgraw-hill.com

[Arnold]

The Java Programming Language, by Ken Arnold and James Gosling, Third Edition.
Addison-Wesley, 2000, 0-201-70433-1.

[Bell]

Java for Students, by Douglas Bell and Mike Parr.
Prentice Hall, 1998, 0-13-858440-0.

[Boone]

Java Essentials for C and C++ Programmers, by Barry Boone.
Addison-Wesley, 1998, 0-201-47946-X.

[Budd]

Classic Data Structure, by Timothy Budd.

Addison-Wesley, 2001,0-201-70002-6.

[Campione]

The Java Tutorial, by Mary Campione and Kathy Walrath.
Addison-Wesley, 1998,0-201-63454-6.

[Carrano]

Data Abstraction and Problem Solving with Java, by Frank Carrano and Janet Prichard.
Addison-Wesley, 2001,0-201-70220-7.

[Chan1]

The Java Class Libraries, Second Edition, Volume 1, by Patrick Chan, Rosanna Lee, and D. Kramer.
Addison-Wesley, 1998,0-201-31002-3.

[Chan2]

The Java Class Libraries, Second Edition, Volume 2, by Patrick Chan and Rosanna Lee
Addison-Wesley, 1998,0-201-31003-1.

[Chan3]

The Java Developers ALMANAC 2000, by Patrick Chan.
Addison-Wesley, 2000,0-201-43299-4.

[Daconta]

Java for C/C++ Programmers, by Michael C. Daconta.
John Wiley & Sons, 1996, 0-471-15324-9.

[Deitel]

Java How to Program, by H. M. Deitel and P. J. Deitel
Prentice-Hall, 1997, 0-13-263401-5.

[Goodrich]

Data Structures and Algorithms in Java, Second Edition, by Michael Goodrich and Roberto Tamassia.
John Wiley & Sons, 2001, 0-471-38367-8.

标准分享网 www.bzfxw.com 免费下载

[Gosling]

The Java Language Specification, Second Edition, by James Gosling, et al.
Addison-Wesley, 2000,0-201-31008-2.

[Grand1]

Java Language Reference, by Mark Grand.
O'Reilly & Associates, 1997, 1-56592-326-X.

[Grand2]

Java Fundamental Class Reference, by Mark Grand and Jonathan Knudsen.
O'Reilly & Associates, 1997, 1-56592-241-7.

[Harary]

Graph Theory, by Frank Harary.
Perseus Books, 1969, 0-201-410033-8.

[Horstmann]

Computing Concepts with Java Essentials, by Cay S. Horstmann.
John Wiley & Sons, 1998, 0-471-17223-5.

[Hubbard]

Schaum's Outline of Programming with Java, by John R. Hubbard.
McGraw-Hill, 1999,0-07-134210-9.

[Kamin]

An Introduction to Computer Science Using Java, by S. N. Kamin, M. D. Mickunas, and E. M. Reingold.
WCB/McGraw-Hill, 1998, 0-07-034224-5.

[Lewis]

Java Software Solutions, Foundations of Program Design, by John Lewis and William Loftus.
Addison-Wesley, 1998,0-57164-1.

[Liang]

An Introduction to Java Programming, by Y. Daniel Liang.
Que Education & Training, 1998, 1-57576-548-9.

[Presis]

Data Structures and Algorithm with Object-Oriented Design Patterns in Java, by Bruno R. Presis.
John Wiley & Sons, 2000, 0-471-34613-6.

[Unicode]

The Unicode Standard, Version 2.0, by The Unicode Consortium.
Addison-Wesley, 1996, 0-201-48345-9.

[Wilson]

Graphs: An Introductory Approach, by Robin J. Wilson and John J. Watkins.
John Wiley & Sons, 1990, 0-471-61554-4.

[Zukowski]

Java AWT Reference, by John Zukowski.

O'Reilly & Associates, 1997, 1-56592-240-9.

标准分享网 www.bzfxw.com 免费下载

译 后 记

Pascal 语言的发明人，瑞士著名计算机科学家 Niklaus Wirth 曾给出“程序设计＝数据结构＋算法”的论断，足见数据结构在计算机科学中的重要作用，因此，也就不难理解数据结构课程在计算机专业课程设置中的核心作用。数据结构是众多在校大学生、有志于进行程序设计的青年学子所必须掌握的。

作为网络环境下体现软件工程和面向对象理念的编程语言，Java 正得到越来越广泛的使用。Java 语言具有简单、面向对象、分布式、解释型、健壮性、安全性、可移植性、多线程和动态性等诸多特点。特别是在网络技术飞速发展的今天，它与 Internet/Web 技术紧密结合，使得更多的人可以方便地使用网络计算资源。可以这样认为，学习并且熟练掌握 Java 语言程序设计基础，不但是计算机专业人员，而且也是众多 Internet/Web 的使用者的需要。

由 Hubbard 编写的《数据结构习题与解答》，一改目前市面上数据结构与 Java 语言教材在内容上相互分离的状况，系统、全面地讲述了如何用 Java 语言来实现各种主要的数据结构，更重要的是，全书使用了大量的源程序，从一个程序员的视角重新诠释了数据结构及其 Java 实现，是学习和掌握数据结构及 Java 语言的一本不可多得的好书。

在教学和科研实践中，我们亲自体会到了本书的价值所在，也深感广大学生需要这种类型的教材和辅导材料，以便尽快掌握数据结构的基础知识和 Java 语言这一编程利器。愿《数据结构习题与解答》一书的翻译出版，能满足这一需要，并为广大在校学生、计算机工程技术人员和编程爱好者所喜爱。

阳国贵主持了本书的翻译、统稿和审校过程，并执笔翻译了前九章。书的后七章和附录部分由周斌和刘亚萍执笔翻译。

在本书的翻译过程中，译者参照了该书的 WWW 主页中的勘误表，对书中的疏漏之处进行了更正。此外，对一些笔误和排版错误也进行了校正。

限于译者水平，译文中难免存在疏漏和错误，欢迎批评指正。

译者

2001 年 10 月于国防科技大学